Proceedings Paper:
Begum, Marjahan, Crossley, Julia, Strömbäck, Filip et al. (2025) A Pedagogical Framework for Developing Abstraction Skills. In: ITiCSE 2024:Proceedings of the 2024 on Innovation and Technology in Computer Science Education V. 2. Annual Conference on Innovation & Technology in Computer Science Education. ACM, NY, pp. 258-299.

https://doi.org/10.1145/3689187.3709613

A Pedagogical Framework for Developing Abstraction Skills

Marjahan Begum*
University of Nottingham
Nottingham, United Kingdom
City St George's, University of London
London, United Kingdom
marjahan.begum@nottingham.ac.uk
marjahan.begum@city.ac.uk

Julia Crossley*
City St George's, University of London
London, United Kingdom
julia.crossley.2@city.ac.uk

Filip Strömbäck*
Linköping University
Linköping, Sweden
filip.stromback@liu.se

Eleni Akrida
Durham University
Durham, United Kingdom
eleni.akrida@durham.ac.uk

Isaac Alpizar-Chacon
Utrecht University
Utrecht, The Netherlands
i.alpizarchacon@uu.nl

Abigail Evans
University of York
York, United Kingdom
abi.evans@york.ac.uk

Joshua B. Gross
California State University Monterey Bay
Seaside, CA, United States of America
jgross@csumb.edu

Pontus Haglund
Linköping University
Linköping, Sweden
pontus.haglund@liu.se

Violetta Lonati
Università degli Studi di Milano
Lab. CINI "Informatica e Scuola"
Milan, Italy
lonati@di.unimi.it

Chandrika Satyavolu
North Carolina State University
Raleigh, NC, United States of America
jsatyav@ncsu.edu

Sverrir Thorgeirsson
ETH Zürich
Zürich, Switzerland
sverrir.thorgeirsson@inf.ethz.ch

Abstract

Abstraction is a fundamental yet challenging skill to teach and learn in Computer Science education. Traditional frameworks of abstraction and concept formation often emphasize understanding an abstraction over its application, the latter being critical for practical Computer Science. Additionally, a common issue in education is when students may understand a concept in a classroom or a very specific setting but struggle to apply it outside of that context. In response, we present here a novel pedagogical framework designed to enhance both the development and application of abstraction skills in diverse educational contexts within the field of Computer Science. Our framework synthesizes common themes from existing models while introducing a new dimension focused explicitly on the actionable development of abstraction skills. Educators can adapt the framework to various educational contexts to support development of students' abstraction skills. Our framework was iteratively developed through a combination of theoretical analysis and reflective practice across multiple teaching contexts. We demonstrate the suitability of the framework by applying it to various case studies, demonstrating its broad applicability and practical utility. By offering a flexible yet comprehensive structure, our framework enables educators to effectively organize and deliver educational content, guiding students from abstract theoretical concepts to their practical application in Computer Science.

CCS Concepts

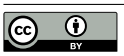
• **Social and Professional topics** → **Computing Education**.

Keywords

abstraction, pedagogy, CS1 to CS3, educational frameworks, computational thinking, algorithmic thinking, inferences, abstraction skills, cognitive models, recursion, pointers, data structures, concurrency, game theory

ACM Reference Format:
Marjahan Begum, Julia Crossley, Filip Strömbäck, Eleni Akrida, Isaac Alpizar-Chacon, Abigail Evans, Joshua B. Gross, Pontus Haglund, Violetta Lonati, Chandrika Satyavolu, and Sverrir Thorgeirsson. 2024. A Pedagogical Framework for Developing Abstraction Skills. In *2024 Working Group Reports on Innovation and Technology in Computer Science Education (ITiCSE-WGR 2024), July 8–10, 2024, Milan, Italy*. ACM, New York, NY, USA, 42 pages. https://doi.org/10.1145/3689187.3709613

*Leader



ITiCSE-WGR 2024, Milan, Italy

ACM ISBN 979-8-4007-1208-1/24/07
https://doi.org/10.1145/3689187.3709613

1 Introduction

Abstraction is a key concept and skill in computer science. It serves several purposes, such as enabling focus on essential information while omitting unnecessary details, or generalizing disparate information to an expression that is more descriptive. In Computer Science Education, students' abstraction skills play a crucial role in

shaping the way they understand and interact with computational concepts. Understanding how abstraction is conceptualized on an intricate scale can inform how it is taught and assessed.

Abstraction and its challenges for teaching and learning has been an area of interest in a range of fields, leading to theories on the development of broad cognitive abilities [13, 65] and models of abstract concept formation in specific domains, such as mathematics [75, 84]. These theories and models have informed frameworks and guidelines for teaching abstraction in mathematics and computer science [2, 6, 29, 64]. Here, we define a *systematization* to be an organized system of processes and objects, intending to describe or define rules (in our context, for the development or assessment of abstraction skills). In these terms, a *framework* can be considered to be a kind of systematization, and in particular a set of predefined rules, guidelines, and/or components that is used to teach and/or understand abstraction. A *model* can also be viewed as a kind of systematization, that is a representation that helps in understanding – a simplified version of reality that captures the essentials. Finally, we also consider a *taxonomy* to be a kind of systematization, and in particular an organisation of objects of the same type into subtypes. In our paper, we build upon the existing systematizations in the literature with a novel framework that is separate from others in two key ways: first, it is widely applicable within the CS domain, and second, its theory can be applied in actionable, meaningful ways by practitioners who are not familiar with the relevant theory. To demonstrate this, we motivate our framework and offer a thorough description of learning contexts where the framework can be applied.

Most of the above literature on abstraction focuses on concept formation, that is, on building knowledge of or about abstract notions. Relevant alongside knowledge, however, are skills and competencies, which describe what students can do and not just what they know. Our framework aims to provide specifically for educators with guidelines for supporting the development of abstraction skills in their students.

To design our framework, we carried out inductive categorization of the existing systematizations to synthesize their common characteristics and organize them according to their goals and application. Simultaneously, we critically reflected on our own teaching practices related to abstraction and how models of abstraction could be applied in our contexts. The final framework is the product of an iterative process combining the two streams of work to ensure that it is grounded in theory and relevant to practice.

The paper is organized as follows. Section 2 reviews the literature about learning and teaching abstraction in CS education and other related disciplines. In Section 3 we illustrate the methods we used to develop and evaluate our framework, starting from the analysis of the systematizations discussed in the literature. In Section 4 we present the results of our analysis, which form the theoretical foundation for our framework. These results include: a description of the abstraction skills that are relevant to CS as we define them, a categorization of the systematizations (models, pedagogic frameworks, tools, etc.) from the literature on abstraction and a synthesis of the contributions within each category. Section 4 also presents an initial version of our framework and its example application to recursion. In Section 5 we apply our framework to a wider variety of educational contexts. We further discuss the case studies and limitations of our framework in Section 6 and we conclude with a summary description of the framework and how educators can use and benefit from it in Section 7. Finally, Section 8 draws overall conclusions on this work.

2 Background

In this section we first provide a brief overview of two widely used general systematizations, then systematizations of abstraction, finally review literature of how abstraction is taught.

2.1 General Systematizations

We first consider existing systematizations that may offer insight into how our work can prove actionable for CS education researchers. Thus, we briefly present literature relating to Bloom's taxonomy and the SOLO (structure of observed learning outcomes) taxonomy, two systematizations which are widely used by practitioners in CS eduction [27].

Arguably the best-known systematization of teaching and learning is Bloom's taxonomy [5, 9]. This taxonomy [9] classified educational objectives in the cognitive domain from the concrete to the abstract, or from the simple to the complex. The revised Bloom's taxonomy [5] presents the domain in the form of a matrix, focusing on describing students cognitive processes. Unfortunately, Bloom's taxonomy has proven sufficiently broad and informal to be problematic when applied CS education; researchers frequently struggled to identify where in the taxonomy certain work could be placed [55]. Further analysis demonstrated structural issues with the taxonomy as a model for classifying work, since both the original taxonomy and the updated taxonomy "fail in providing non-ambiguous definitions of cognitive processes and their categories" [91]. From this, Bloom's taxonomy can be considered insufficient to explore the specific problem of teaching and learning abstraction in computer science. A verb list specific to computing developed by the ACM Committee on Computing Education in Community Colleges focuses on the production of technical artifacts, rather than the aspects of computer science more strongly related to abstraction [1].

Rather than serve as a taxonomy, the SOLO model is a model for taxonomies, and educators and instructional designers use the model to create a taxonomic progression of a learning outcome. Unlike Bloom's taxonomy, the SOLO taxonomy model focuses on the internal understanding of the learner, and attempts to model that understanding with specifically-designed learning tasks that move the student to a deeper level of understanding[8]. As such, this model is relevant to the framework we develop. SOLO has been applied in CS education research, particularly in assessing how novice programmers design their solutions [14, 34, 44]. However, while it does reflect on internal knowledge state, SOLO is not specific to either abstraction or computer science. It is also a prescriptive model through which educators are encouraged to build a complete taxonomic progression. The framework we propose is intended to serve as a guide for educators looking to improve abstraction learning in relevant contexts, rather than a formal model for designing learning progressions.

2.2 Models and Frameworks of Abstraction

Models of abstraction and concept formation have been proposed in research fields including cognitive science, mathematics education, and computer science education. In this section, we give a brief overview of how key models of abstraction skills have evolved and how they have been incorporated into frameworks for teaching specific and general abstraction skills. This section provides limited details on the models themselves; more details and analysis can be found in Section 4.3.

Three researchers and theorists have proposed models of abstraction that have been influential in later work on the subject: pyschologists Jerome Bruner and Jean Piaget, and mathematics education researcher Anna Sfard. For each of these models, abstraction is a three-stage process.

Bruner's [13] model of the development of internal (cognitive) representations of concepts and processes begins with the *enactive* stage, in which thought is grounded in physical actions and concrete processes. Next, in the *iconic* stage, visual representations can be used to summarise actions and objects. Finally, in the *symbolic* stage, the thinker is able to reason using language independently of actions and images. Bruner's model is not specific to abstract concepts, but abstraction ability is embedded in the iconic and symbolic stages through the use of representations that are increasingly independent from the original concrete actions and objects. Fyfe et al. [28] incorporated Bruner's model into their "concreteness fading" framework, which advocates presenting concepts using teaching materials corresponding to each of Bruner's stages in order to guide students from concrete to abstract understanding.

Piaget [65], a contemporary of Bruner, described three types of abstraction: *empirical*, *pseudo-empirical*, and *reflective*. In empirical abstraction, knowledge of an object's properties are derived directly from concrete objects. In reflective abstraction, the learner is able to generalise beyond specific concrete objects. Pseudo-empirical abstraction describes the beginnings of shift from empirical to reflective abstraction. The APOS (Action, Process, Object, Schema) model [26] of abstraction in advanced mathematical thinking builds on Piaget's concept of reflective abstraction.

Sfard [75] identifies three ordered stages of abstract concept development: *interiorization*, *condensation*, and *reification*. In the interiorization stage, the learner becomes familiar with the process at the heart of the abstract content. In the condensation stage, the learner is able to think of the process as a complete entity with an output, although the process is still central to the learner's understanding. Reification occurs when the learner understands the abstract concept as an independent object separate from the underlying process. Interiorization and condensation take time to develop but reification is instantaneous [75, p. 20]. Key to Sfard's model is the notion that an abstract concept can be both an object and a process. Aharoni [2] simplified Sfard's model and applied it to the analysis of computer science undergraduates' thinking processes when working with data structures.

Hazzan [37] developed a framework to explain algebra students' tendency to reduce the level of abstraction of concepts they are learning in order to work at a more concrete level. The framework builds on the ideas of Piaget, Sfard, and the APOS model and has also been applied in computer science education [33, 38, 40, 41]. Ginat and Blau [33] used the framework to develop guidelines for teaching abstraction in algorithm design; Hill et al. [41] used Hazzan's work to inform their own scales of abstraction ability; and Hazzan's framework informs Perrenet, Groote, and Kaasenbrood's [64] PGK hierachy, which describes algorithmic thinking at four levels of abstraction (execution, program, object, and problem).

The PGK hierarchy has also proved influential in computer science education research. For example, Izu [43] draws on PGK and Hazzan's work for their definition of abstraction; and Armoni [6] has developed a framework for teaching abstraction based on the PGK hierarchy.

The Block Model [71] of program comprehension consists of three dimensions (text surface, program execution, and function/purpose) of a program and four levels of zooming (atom, block, relational, and macro) that apply to each dimension. Although the model does not share the same theoretical roots as the models and frameworks above, nor is it explicitly a model of abstraction, it is included here because it concerns how students build an "abstract and general model" of a program [71, p. 150].

In the next section, we discuss how these models and frameworks have been applied to teach abstraction and abstract concepts in computer science classrooms.

2.3 How Abstraction is Taught

Literature on abstraction in the CS classroom can be roughly grouped into three parts: assessing abstraction, teaching the role of abstraction, and teaching to trigger abstraction [56]. Given that the scope of our work is developing abstraction skills, we review here only the contributions from the latter group following the structure in [56] and adding some recent works that have appeared since.

Teaching to trigger abstraction aims to develop abstraction skills implicitly by having students engage in tasks that require them to use abstraction [56]. Pattern-oriented instruction (POI) is one example of such a pedagogical approach, which concerns the abstraction processes when solving algorithmic problems [57]. Muller and Haberman [57] specifies three processes of note concerning instructional design and students' problem-solving behaviors: pattern recognition, chunking, and identification of problem structures. Their [57] POI approach supported students' abstraction processes and enhanced students' problem-solving ability.

Further, when developing abstract concepts, dealing with multiple representations of a phenomenon can be important [56]. While multiple representations of complex scientific concepts are commonplace, they are not always effective [3]. Ainsworth [3] found that, for multiple representations to be effective, teachers need to help students relate multiple representations to concepts. Gautam et al. [30] found that multiple representations require students to move between different representations of a concept. They recommend that the friction that occurs when students explore different representations is useful since it requires students to revisit their understanding of a concept.

Another approach to triggering abstraction is the exploration of artifacts by students [56], essentially going through three stages to understand a program: using the program, experimenting by modifying features, and then creating new programs to achieve some goal (use-modify-create) [56]. Lee et al. [50] found that abstraction

is not taught to students but rather developed when students work with creating, meshing well with the approach of use-modify-create.

When teaching to trigger abstraction, it can be useful to evaluate to which extent abstraction is represented in tasks that students engage in. Cutts et al. [22] created the abstraction transition taxonomy which can be used to assess tasks that teachers have their students engage in. The taxonomy measures the extent to which abstraction is reflected in the task, and has three levels that are related to language used; essentially, students are asked questions, and these questions are expressed at three levels of abstraction that relate to language: English, CS Speak, and Code. Cutts et al. [22] focused on the transitions between the beginning and the end of an assignment and did not cover any intermediate steps taken, i.e., a question that transitioned from level 3 to level 1 could be: given this code, choose an English description that describes the goal.

The pedagogical approaches described above focus on triggering abstraction but do not but do not consider other factors, such as stakeholders, responsibility and the impact of systems created. Thus, it is worth noting that the dominant role of abstraction in CS education has received criticism [54, 68, 72]. Malazita and Resetar [54] argue that abstraction as a practice encourages students to view the social and political context of the systems they create as separate from their technical implementation, thereby discouraging criticism of the impacts of technical systems. They propose an alternative approach to teaching CS1, called 'Critical CS1', which embeds discussion and criticism of the impacts of technical systems alongside the introduction of foundational programming concepts. Focusing on AI ethics education, Raji et al. [68] argue that current approaches center around abstracting complex social issues into more approachable technical problems, thereby excluding more nuanced perspectives. Selbst et al. [72] refer to this as an "abstraction trap", in which the standard practices of computer scientists can lead to proposed solutions that are inherently ineffective. Raji et al. [68] suggest a collaborative approach to education that encourages engagement with other disciplines' ways of knowing and doing to temper the CS tendency to abstract away details. While we find these perspectives to be valuable for the sustainability of CS eduction, we focus on the cognitive aspects of understanding and applying abstraction rather than larger questions of the role of abstraction in the CS curriculum as a whole.

2.4 Generative AI

Since the release of Codex[1] and ChatGPT[2], there has been significant interest in exploring the use of Generative Artificial Intelligence (GenAI) in computing education [66]. This interest has also led to the development of tools that support abstraction skills to varying degrees. Liu et al. [52] explored abstraction matching, the process of aligning a user's intent with a system's capabilities through an appropriate level of abstraction. They proposed grounded abstraction matching, which converts system-generated actions (e.g., code) into editable, naturalistic examples that help users reliably replicate actions. Tested in spreadsheet data analysis, the approach used a system that translates natural language queries into Python code via Codex. A user study found that grounded examples improved users' mental models and confidence over time.

Another abstraction skill explored with the help of GenAI is the ability to transition between different levels of code representation. Denny et al. [23] introduced a novel programming exercise called "Prompt Problems," designed to teach students how to create effective prompts for AI-based code-generation tools. In these exercises, students are presented with a problem illustrating how input values should be transformed into output and tasked with crafting a prompt that guides a language model to generate the necessary code. The authors developed a tool to deliver these prompt problems, which was tested in two Python-based courses. Students appreciated the opportunity to learn new programming constructs, analyze the structure and syntax of generated code, and improve their problem-solving skills. Another approach, explored by Smith et al. [77], involves using GenAI for "Explain in Plain English" questions. These exercises provide students with a piece of code and ask them to describe its functionality in natural language [58], helping train and assess their ability to understand and articulate the purpose of code. The study found that students were generally successful in completing these prompts, effectively conveying the code's high-level purpose without focusing on implementation details. However, students also expressed critical views about GenAI in general.

Finally, GenAI can assist students in understanding (abstract) computer science concepts [24]. Recent studies indicate that GenAI is increasingly becoming a preferred help-seeking tool for many students [46]. However, integrating GenAI into computing education presents significant challenges. Prather et al. [67] found that struggling students may encounter difficulties when using GenAI, and some students have expressed concerns about overreliance on these tools in various learning situations [90, 96]. While there are no specific studies on the effectiveness of GenAI in teaching abstraction skills, research in related areas (e.g., programming error messages) shows that human explanations continue to outperform GenAI [70]. GenAI could be a supportive tool for teaching abstraction skills, particularly when integrated into a broader educational framework or guided by a teacher, rather than being used as a standalone approach.

3 Method

The work began with identifying relevant literature, drawing on the prior work of group members. The working group includes established academics with a broad perspective on the relevant academic literature [56, 61] and researchers actively exploring abstraction in Computer Science Education [18–20]. This initial body of work spanned diverse disciplines, including cognitive science, mathematics education, and computer science education, focusing on both theoretical and empirical studies on abstraction [18–20, 56, 61]. Members took responsibility for expanding this body of literature through a snowballing approach [95], where abstracts of newly identified papers were reviewed based on whether they explored abstraction theoretically, empirically, or both. Relevant papers were then incorporated into the background literature presented in Section 2. It is important to note that while the present work is not a systematic literature review, the members' extensive prior work

[1] https://openai.com/index/openai-codex
[2] https://openai.com/chatgpt/overview/

in this area provided confidence that the identified literature offered comprehensive insights into abstraction, both as theoretical groundwork and empirical application.

We then organized the group work with two parallel groups based on research interests, areas of expertise, and experience in teaching abstraction skills. We refer to these groups as the "Theory Group" and the "Context Group": members with strengths in theoretical work formed the Theory Group, while those primarily focused on teaching practices joined the Context Group.

The Theory Group began by analyzing existing theoretical and empirical work on abstractions and proposing an initial version of the framework (see Section 4). Simultaneously, the Context Group collaborated with the Theory Group to identify relevant contexts and explore ways to integrate the initial framework into practice. This collaboration prompted further discussions within the Theory Group about aspects of the framework that could or could not be adapted to learning situations. Through multiple iterations of "back-and-forth" exchanges, the groups refined the framework and contexts alongside one another, ultimately resulting in the final framework described in Section 7, which was aligned by both groups.

To summarize, the Theory Group followed an inductive process to: identify key abstraction skills, define categories to group all systematizations, establish inclusion and exclusion criteria for systematizations, and identify stages of development, culminating in the initial framework (presented in Section 4). Throughout these steps, the Context Group played an integral role. They began by familiarizing themselves with the original systematizations and describing their learning situations in alignment with the key systematizations. They actively participated in discussions at each stage, providing feedback that the Theory Group used to refine the framework. For example, the Context Group evaluated whether the abstraction skills identified by the Theory Group were reflective of those observed in their learning contexts. As the Theory Group employed an inductive approach, allowing the Context Group to adopt a more deductive stance, using the Theory Group's findings and evolving framework to inform their analysis and applications, as broadly framed in [85]. Further details of the methodology are provided in Section 3.1 for the Theory Group and Section 3.2 for the Context Group.

3.1 Work of the Theory Group

As introduced earlier in this section, the goal of the Theory Group was to synthesize prior research on abstraction into a coherent structure that would serve as the foundation for developing the framework. This framework would then guide educators and researchers in analyzing and improving instruction on abstraction.

Given the overarching goal of creating a framework for developing abstraction skills, the Theory Group focused on the following key areas, refined through extensive discussions and analysis of the literature:

- identifying and agreeing on the computational abstraction skills CS educators aim to foster in students;
- categorizing existing systematizations of abstraction;
- establishing criteria for including or excluding systematizations in the analysis;
- defining stages of abstraction skill development;
- setting criteria for an effective and purposeful pedagogical framework.

Based on the literature, the Theory Group first proposed a set of essential abstraction skills that CS educators aim to cultivate in students (Section 4.2) as a means of identifying and focusing on relevant systematizations. Concurrently, the group inductively identified categories –such as models of concept formation, taxonomies, and pedagogical frameworks– for organizing these relevant systematizations found in the literature. Systematizations were then categorized accordingly (Section 4.3). These decisions were revisited and refined iteratively over several meetings to ensure consistency and alignment with the group's goals.

The theoretical work led to the establishment of criteria for including or excluding systematizations (Section 4.3.1). These criteria were used to filter the identified systematizations, enabling the group to discern common themes among the selected works. These themes, combined with the identified abstraction skills, led to the definition of stages for the development of abstraction skills and eventually formed the foundation of the initial framework.

Further the categorizations of the data from Section 4.3 led to the extrapolation of a common process within the categories of systematizations; these can be found in Table 3.

3.2 Work of the Context Group

As introduced earlier, the Context Group played a crucial role in the development of the initial framework and its application on concrete learning situations. Their first step was to select examples of learning activities where developing abstraction skills is fundamental. These examples, referred to as *contexts*, were analyzed by using existing systematizations to help familiarize members with both the contexts and the systematizations. Once the initial framework was established, the contexts were re-evaluated and aligned with the framework, as shown in Table 4. The goal was to apply the initial framework to existing practices and provide practical, worked examples of its application.

The application of the framework varied depending on the specific context. For instance, in Context 2, the introductory nature of the material meant that fully transitioning between abstraction levels was not feasible, and there was limited scope for creating abstractions. Instead, the focus was primarily on understanding in Stage 1 and Stage 2 of the framework.

Applying the framework to real-world contexts also provided valuable insights into how abstraction skills could be taught under different constraints. For example, mapping abstraction skills to the stages identified in the framework highlighted specific challenges and opportunities.

These insights were used to iteratively refine the framework, enhancing the clarity and detail of each stage and specifying distinctions among the skills of identifying, moving, and creating as they apply to different stages. Group discussions enabled the framework's structure to evolve until consensus was reached. This process also addressed cases where the framework was challenging to apply or where additional details were needed. Moreover, the insights contributed practical examples of activities educators could

use to help students develop specific abstraction skills at particular stages.

Finally, additional insights from applying the framework to specific learning situations were captured. These were summarized as practical tips for educators, offering guidance on structuring lessons to teach multiple concepts or abstraction skills concurrently.

3.3 Interactions Between the Two Parallel Groups

The candidate abstraction skills –identifying, moving, and creating– were presented by the Theory Group and discussed in detail with the Context Group. These discussions focused on how these abstraction skills could be encouraged within the learning situations encountered by the Context Group. For example, what does the skill of identifying entail when helping students understand machine-level memory representations in C for a simple variable? At what stage are students able to apply this understanding effectively? How can students be guided to transition between abstraction levels (Section 5)? Similarly, specific clarifications and refinements were incorporated based on the diverse teaching contexts analyzed in the Context Group. These discussions were partially informed by members' teaching experiences, as outlined in Section 5.

This collaborative approach drew upon the expertise of all members in the working group, offering the additional benefit of accommodating diverse perspectives. For instance, in introductory programming (see Context 1 in Section 5.1), the skill of creating abstractions may not be essential, whereas in concurrent programming (see Context 3 in Section 5.3), it becomes critical.

The insights gained from these discussions shaped the initial framework, presented in Section 4. This preliminary framework was then applied and reflected upon by the Context Group in their respective teaching scenarios. Their feedback and observations contributed to further refining the framework. The final version of the framework is presented in Section 7.

4 Results: Design of the Initial Framework

4.1 Our Views

In Section 2.2 we have described systematizations linked to abstraction that differ in aim and type; some describe abstraction processes, some describe abstraction skills and some describe levels of abstraction. The latter category may describe different 'measures' of abstraction level, such as Hazzan's mental processes linked to reducing abstraction level [37], or a 'spectrum' of abstraction levels such as the PGK hierarchy [64].

As instructors, our aims are to encourage conceptual understanding by guiding students to develop abstraction skills and apply this understanding to computational problems. We therefore need to consider systematization that describe how students form conceptual understanding in order to guide students through mental processes that link well to abstraction skills, while considering how to best help students cement their understanding.

A good pedagogical framework for our purposes therefore needs to:

- use the conceptual stages in systematizations that describe how students develop understanding as thresholds;
- guide students through these stages by making them use mental processes that require abstraction skills;
- be specific enough that it is actionable within a Computer Science education context by helping instructors understand the what, how, and why;
- be general enough that it is as widely applicable as possible to a Computer Science context.

The next section develops a classification of the selected systematizations, identifies common features from each, and then integrates these features.

4.2 Definitions for Abstraction Skills

4.2.1 Measures of Abstraction Levels. Abstraction levels are degrees that describe how abstract something is. Different views or measures exist to describe different abstraction levels along a spectrum of higher (more abstract) to lower (more concrete).

In the **process-object duality**, as discussed by Sfard [75], a new concept at a lower level is perceived as a mechanical process (operational conception) that acts on other objects (structural conception), such as an algorithm for performing a mathematical operation. Once a higher abstraction level is reached, the new concept is perceived as an object that can act on other objects, with the process fading in importance to the student's mental model. Seeing an entity as a process implies "regarding it as a potential rather than an actual entity, which comes into existence upon request in a sequence of actions". Conversely, an entity as an object means "being capable of referring to it as if it was a real thing and to recognize the idea 'at a glance' and to manipulate it as a whole, without going into details" [75].

Hazzan [37] examines three interpretations for levels of abstraction discussed in the literature, while referring to students' tendency to work on a lower level of abstraction (more concrete). The first interpretation focuses on *abstraction level as the quality of the subjective relationships between the object of thought and the thinking person*. The more familiar and connected a person is to an object, the more concrete and less abstract it feels to them. The second interpretation is *abstraction level as a reflection of the process-object duality*, as introduced before. The last interpretation is *abstraction level as a reflection of the degree of complexity of the thought concept*, and involves adding a simplification to new concept, such as by focusing only on a specific case.

In a more applied form, the **PGK-Hierarchy** [63, 64], defines four levels of abstraction for the concept of an algorithm. The lowest level is *execution* (algorithm is a specific run), then *program* (algorithm is a process), moving to the *object* level (algorithm is an object), and finally the highest level is the *problem* level (algorithm is a black box).

In the next subsection, we define generalizations for abstraction *skills*, which incorporate abstraction *levels*. These skills relate to the levels and measures described above.

4.2.2 Abstraction Skills. There is no universally accepted definition of abstraction skills. Some authors extensively describe and categorize abstraction skills, while others do not address them at all. For example, while Statter and Armoni [79] presents two categories (related to PGK-Hierarchy [63, 64]), six operational skills, and four

supporting indicators, Böttcher et al. [11] talks about abstraction as a skill, but does not break it down into parts.

Hill et al. [41] talks about abstraction skill as "cognitive abstraction", but identifies three elements of abstraction: "conceptual abstraction, formal abstraction and descriptive abstraction". Nicholson et al. [62] consider abstraction in terms of five key skills: "considering the context in which the abstraction operates, defining and choosing the right abstraction, working with multiple layers of abstraction simultaneously, considering abstractions critically, and knowing when to use abstraction".

Kramer [48] states that abstraction is a key skill for computing without explicitly listing abstraction skills. He extracts certain important attributes of a general definition of abstraction:

- *The act of withdrawing or removing something, and; The act or process of leaving out of consideration one or more properties of a complex object so as to attend to others. (...)*
- *The process of generalization to identify the common core or essence based on the definitions: The process of formulating general concepts by abstracting common properties of instances, and; A general concept formed by extracting common features from specific examples.*

Finally, Hazzan and Kramer [40] described several indicators of good abstraction skills (as identified by a group of expert educators in software development and computer science):

- *Ability to **invent** new abstractions at different levels of abstraction.*
- *Ability to "**do**" abstraction—to throw away detail, while keeping the essential structure. Also, ability to "do" refinement—to add detail.*
- *Ability to **CREATE** (rather than just assess), from an external knowledge source, different levels of abstraction.*
- *The ability to **maneuver** between abstraction levels, as needed, by adding/removing details. Also, to identify the entities that comprise an abstraction level, and, constructively, detailing or abstracting them in order to achieve the desired level of abstraction.*
- *… the ability to **devise** abstractions at various levels.*

Building on these indicators by Aharoni [2], Dubinski [26], Hazzan [37], Hazzan and Kramer [40], Perrenet and Kaasenbrood [64] and others, we generalize abstraction skills into three core competencies: *identifying*, *moving*, and *creating*. Each competence is defined in turn with pertinent literature.

Identifying. It is the ability to recognize a specific object or situation and classify it as an instance of a known abstract concept. This skill necessarily requires the ability to disregard non-essential elements while preserving the fundamental structure. In a study by Perrenet et al. [63], all participating instructors agreed with a definition of abstraction as either *disregarding detail* or *condensation*. This connects to Sfard's condensation stage [75], which can be interpreted as making judgments on what information can be reduced, emphasizing the capability of "[...] 'squeezing' lengthy sequences of operations into more manageable units" [75, p.19] and "[...] thinking about a given process as a whole, without feeling an urge to go into details" [75, p.19]. An example of this skill would be the recognition that an object/situation could be modeled using an abstract concept (e.g., a particular set of data could be modeled as a graph).

Moving. It is the capacity to navigate between different levels of abstraction to achieve the desired outcome; this can be evidenced through artifacts such as code or drawings. Moving involves adjusting detail as required and identifying and detailing the entities within a given abstraction level. It can also be viewed as an oscillation between lower and higher levels of abstraction. In Perrenet et al. [63]'s experiment, teachers highlighted making abstraction-level distinctions as a part of abstraction. A higher capability of Sfard's condensation stage [75] can be seen as the "[...] growing easiness to alternate between different representations of the concept" [75, p. 19]. The first two steps in Armoni [6]'s framework also emphasize the moving skill: "differentiate the levels" [6, p. 273] and "move freely and consciously between the levels" [6, p. 273]. Finally, Fyfe's concreteness fading framework [28] is based on multiple representations of concepts moving from concrete materials toward more abstract ones. An example of this skill involves moving between specification, pseudo-code, and implementation for a specific problem.

Creating. It is the ability to generate new levels of abstraction and consciously understand and manipulate these levels (awareness). Armoni [6], while referring to Hazzan [39], mentions that it is not enough for students "to perform abstraction and move between levels of abstractions, they must also be aware of it" [6, p. 271]. In Armoni [6]'s framework, the last step refers to the ability to "use successive refinements of abstractions" [6, p.273]. Sfard's reification stage [75] is when " a process solidifies into object, into a static structure" [75, p. 20]. Similarly, in the APOS's model [17], a higher level of abstraction is constructed when "the individual becomes aware of the process as a totality and realizes that transformations can act on it" [17, p. 4]. For example, when designing the ER model for a database or defining attributes of classes and their relation to other classes in OO design.

Separately, we have identified two *outcome skills* that can be perceived as the goal of the other three skills for teaching and learning concepts, depending on the desired outcome—the understanding or application of concepts.

Understanding. It is when an abstract idea or concept forms in a student's mind, it becomes an object that they can build upon, following the process-object duality described by Sfard [75]. This is what students experience in their minds after the *a-ha* moment in Sfard's model. For example, understanding *iteration* as a programming construct occurs when a student can think of iteration as a tool to be applied to solve a problem, without the need to think of it operationally as a repetition of actions, tracing/executing it, or referring necessarily to its syntax in a specific language. This is a necessary step toward helping students understand theoretical models of computation (i.e., programs can be built using sequence, iteration and selection only).

Applying. It is when an abstract idea or concept is used for a specific goal. Again considering the concept of *iteration*, it can be applied in many ways, for example writing or understanding a

program that uses loops, using a loop to sum the number of items in a series, implementing a loop in a specific programming language, or understanding where a loop is useful for some programming problem.

Figure 1 summarizes our model of abstraction and outcome skills.

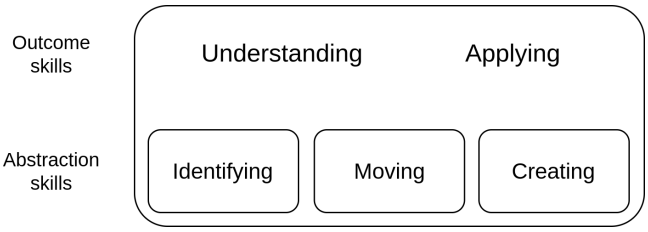


Figure 1: Generalization of abstraction skills.

4.3 Categorization of Systematizations

We divide the systematization into three broad categories, keeping in mind that there may be some overlap between them. These can be found in Table 1.

Table 1: Our categories of systematization.

Models of concept formation [2, 13, 26, 65, 75, 76]	These have the rather ambitious aim of trying to model the process by which understanding is formed; they are meant to describe a 'natural', 'necessary' progression in developing understanding.
Taxonomies of abstraction skills, processes or levels [38, 40, 41, 64]	These classifications aim to describe how to measure attributes of abstraction; for example, Hazzan describes three distinct variables by which a level of abstraction can be measured [37]; the PGK hierarchy defines levels of abstraction of the concept of an algorithm [64].
Pedagogical frameworks [29, 71]	These are used to convey or assess knowledge or skills (rather than model how students form understanding). For example, *concreteness fading* can help students make inductive inferences, which could use a model of concept formation (e.g., Piaget or Bruner) by gradually abstracting from easily-contextualized concrete observations (i.e., observable in real life) to enable the transfer from empirical to pseudo-empirical abstraction.

Our reasons for dividing up the systematizations into these three categories will become clearer as we go through the analysis. Our process is to determine what the systemazations do and do not have in common.

4.3.1 Inclusion-exclusion. Before we go further into our analysis it is important that we discuss which systematization we have decided to exclude and why.

We have decided to exclude those that describe 'languages', that although they require abstraction skills to understand and use, are not abstraction systematization in themselves; they are representations, such as Java or UML diagrams. Some of these 'languages' could however be used in conjunction with one of the frameworks, such as Fyfe and Nathan [29].

The systematization we have decided to use are influential, in that they have been used as a building block for other research, including other systematization we are examining; and described well enough in their design such that we can make reasonable assumptions in analyzing and applying them in contexts outside what is already in the literature.

4.3.2 Models of Concept Formation. The models of concept formation described in Section 2.2 are summarized in the Table 2.

Aharoni describes the Action - Process - Object framework as a simplification of APOS and process-object duality [2, 26, 94]. They therefore describe a common process.

Piaget's and Bruner's systematization have different purposes and views on the development of abstract thinking. Piaget's model explains observations of how children develop full cognitive capabilities, while Bruner's specifically describes concept formation for the purposes of instruction. Bruner also shows more awareness of context that is local to the thinking person [83]. Despite these and other differences, both models describe a common path that, like APOS, Sfard and Aharoni, can be described in three to four stages.

SOLO [12], although not specific to a single discipline, also follows a path common to the other models in this category.

In each model there is a stage of observation: concrete objects are observed and actions are performed using the objects. In Piaget's setting this could be observing a ball and rolling it around [65]; in a mathematical context this could be moving a point along a path; in a computational one this could be tracing through a block of code. It requires 'direct observation' and description. In Bruner's model, this would begin with a physical action, such as moving and counting physical coins (enactive stage).

The next stage requires some reasoning about the observations; this could be seeing cause and effect, identifying equivalences, or combining observations together for example. In Sfard's model this is where condensation and perhaps some reification happens; in APOS it is understanding actions as processes and linking them to the object [26]. In Bruner's model, the student could recognize that an image of a number of coins is equivalent to physical coins, and they can be counted mentally (iconic stage).

The third stage involves hypothesizing about the observation: this could be inferring how the concrete object would behave in another setting, or how a different concrete object would behave in the same setting, for example. It could also mean 'constructing' something new, using reflections from the other two stages. In Sfard's model this is where the concrete object becomes a concept that the student is able to apply [75]. In Bruner's model, the student could recognize that the digit 4 is equivalent to an image of four coins, and that the digit replaces the need to count (symbolic stage).

In other words, the stages are:

Stage 1: Observe stationary things and actions on them: by this we mean observe concrete objects and how they may change under certain conditions.

Table 2: Models of concept formation.

Piaget [65]	Bruner [13]	Sfard [75]	APOS [17]	Aharoni [2]	SOLO [12]
Empirical abstraction	Enactive stage	Interiorization	Action	Action	Pre-structural and uni-structural
Pseudo-empirical abstraction	Iconic stage	Condensation and reification to object	Process and Object	Process and Object conceived as action	Multi-structural and Relational
Reflective abstraction	Symbolic stage	Concept	Schema	Object conceived as process	Relational and Abstract extension

Stage 2: Make inductive and deductive inferences about the observations: attempt to explain the observed process causing the change; combine processes.

Stage 3: Think hypothetically using the observations: infer behaviour on other unobserved concrete objects or unobserved processes; generalise behaviour to a more abstract level.

The example of a loop is helpful in illustrating this [15]. Stage 1 could be manually tracing through a loop, evaluating test conditions and values of variables as the loop iterates. The student would have reached Stage 2 if they were inferring meaning from Stage 1, without tracing each line; at an advanced Stage 2, they would be meta-tracing. In they Stage 3, they would be hypothesizing on how the loop would impact the execution of a larger program, for instance in creating nested loops or realising using another structure is more efficient.

A further generalization of this common thread is the understanding of the process and the object.

4.3.3 Taxonomies of Abstraction Skills and Levels. These are taxonomies of aspects of abstraction that are of the same type; subtypes are then defined within this.

Hazzan's work on reducing levels of abstraction could conceivably have been categorized as a model of concept formation: the basis for this taxonomy suggests that the reduction of abstraction level is a subconscious process the student goes through as they organize information [37, 43]. However, we consider that the contributions that are most valuable to us in Hazzan's taxonomy are the defining features of abstraction levels. We take it as a given that determinants of abstraction level should be defined by natural cognitive tendencies.

The aspects of abstraction that are in these taxonomies are:

- Variables that measure abstraction level; by this we mean things that change between levels of abstraction such as how removed the representation is from the student's context [37].
- Hierarchies of levels of abstraction relevant to a particular purpose, such as algorithm conception [64] or problem-solving strategies [43].
- "Scales" of abstraction, describing distinct types of abstraction skills [40, 41].

Many of these taxonomies build on each other. In particular, Hazzan and Kramer's work [37, 40] is a building block for the PGK hierarchy [64]. The same can be said of Hill et al.'s triad of abstraction types: conceptual, procedural and descriptive abstraction [41].

Izu's work generalizes a framework designed by Ginat and Blau [33] for a very specific problem. Although Ginat et al.'s framework is useful, and certainly has common points with Hazzan's work, we believe that Izu's taxonomy gives the best balance of describing the 'why', 'what' and 'how'. The same again can be said on Izu's work in developing a taxonomy of abstraction levels for algorithmic problem solving [43].

In our interpretation of Izu's hierarchical abstraction taxonomy [43] of problem solving actions, starting with the lowest level of abstraction, the student:

- Level 0: Uses trial and error to (for example) attempt to test 'all' possible inputs. The caveat here is that their tests may not be exhaustive.
- Level 1: Identifies inputs that could be grouped together (this could also be linked to a less sophisticated stage of Sfard's Sfard [75] condensation stage).
- Level 2: Infers a relationship on the groups of inputs.
- Level 3: Further hypothesizes on relationships between groups of inputs.

We could generalize the applicability of this taxonomy to reasoning on a broader level, where 'inputs' are not parameter values but computational functions that have a known algorithmic purpose.

We find that describing this in terms of inferential reasoning is useful: this does not describe what the student does mechanically, but the conclusions they come to as a result of what they do. This encourages us to remain observant of what the student does with the information available to them, which in turn can help us identify what knowledge the student is secure with.

(1) Observation-induction
(2)(a) Induction
  (b) Deduction
(3) Hypothetico-deduction [21, 65]

We have made a distinction between induction on a static observation and induction on a process. We think it is important to do this so that the important features in the stages identified in Section 4.3.2 are clear here as well.

4.3.4 Frameworks for Pedagogical Practice. These frameworks are easily applicable to pedagogy in that they are designed with a clear stated pedagogical purpose in mind and they are descriptive of

the process an educator should take in achieving this. The PGK hierarchy is not included in this category as, although it is designed with the staggered understanding of an algorithm in mind, we consider it as a taxonomy that *defines* levels of abstraction as objects but does not *describe* how to enable the process of moving between stages. Pedagogical frameworks could have any of the following purposes:

- developing a clearly stated skill, such as the Block model [71];
- transferring knowledge, by describing how to present information to students; an example of this is Fyfe's concreteness fading framework that proposes a method for enabling students to [29]:
  (1) use inductive thinking to generalise an instantiation (or several), such as generalising from concrete values to a variable;
  (2) use a change of representation to understand a more abstract idea or representation. We could see this as moving to a higher level abstraction, where the abstraction level is determined by how close the representation is to the person. In other words, it is consistent with one of Hazzan's measures of abstraction level [37].
- developing schemata that have 'objects' that are close to the context of the student and that allow them to build correct mental models; these are notional machines [78].

4.4 Discussion/Summary of the Systematization

The models of concept formation and taxonomies of abstraction show similar features; these are given in Table 3. This gives us one dimension for the framework we wish to design, which we will later combine with elements of existing pedagogical frameworks given in Section 4.3.4; in this way we will build a two dimensional framework. This process is described in Section 4.4.1; in the meantime we provide a summary of our insights in Table 3.

Table 3: Common features of our categorizations of models of concept formation and taxonomies of abstraction.

Stages through the models and taxonomies
Stage 1: Observe concrete things and actions on them (observation and induction)
Stage 2: Make inductive and deductive inferences about the observations (induction and deduction)
Stage 3: Think hypothetically using the observations (hypothetico-deduction)

4.4.1 Steps for Constructing our Pedagogical Framework. The pedagogical framework for teaching abstraction we wish to design can make use of the common thread we have identified in Table 3 and our analysis of the frameworks for pedagogical practice in Section 4.3.4.

A first step to doing this is to consider how models of concept formation manifest in the existing frameworks: these pedagogical frameworks are informed by cognitive and educational theory; it would therefore be useful to identify how this theory has been used in the construction of our framework.

This would allow us to think about how we can effectively combine the dimensions of stages and the ones of skills in a manner that builds on the approach described in Nakar et al. [59]. Our approach differs in that the two dimensions are treated differently, unlike in Nakar et al.'s work where abstraction levels and problem solving phases are measured using the same scale. We consider that for the purposes of being widely applicable to teaching and learning, the approach and style of description need to be different.

4.5 Initial Version of the Framework

Building on the analysis described in this section, we have designed a three-component pedagogical framework:

(1) meta-abstraction skills—*understanding* and *applying* as outcome skills related to concepts; and *identifying*, *moving*, and *creating* as intricate skills connected to abstractions levels (see Section 4.2.2);
(2) approaches taken to allow for progression in understanding, as indicated by insights from analysing the pedagogical frameworks in the literature (see Sections 4.3.2, 4.3.3, 4.3.4);
(3) stages in developing understanding, as indicated by the common stages in the models of concept formation and taxonomies of abstraction (see Section 4.4).

To use our framework as a pedagogical tool, we propose combining the three components, as seen in Table 4. An example of this is given in Section 4.6. Table 4 summarizes our initial framework as follows:

Stages: the left-hand column represents the inferential stages we described in Table 3;
Understanding and Applying: the second and third columns from the left describe the meta-skills in abstraction discussed in Section 4.2.2; each cell represents a combination of the abstraction skill and the inferential stage;
Identifying, moving and creating: these intricate skills are needed at all points U1 to A3; how this can manifest in practice is explored in Section 4.6 and Section 5.

Table 4: Initial version of our pedagogical framework showing the two outcome skills across the three stages of cognition & the three abstraction skills.

	Understanding (U)	Applying (A)	Skills
Stage 1	U1	A1	Identifying, moving, creating
Stage 2	U2	A2	
Stage 3	U3	A3	

4.6 Example: Applying the Framework to Recursion (Context 0)

We have chosen a simple example of a concept—recursion—to illustrate our framework. Our starting point was a Jupyter Notebook that is used to teach recursion[3] to non-CS students at a large university in the Netherlands. We then explored ways to enhance the content based on the perspectives discussed in Section 4.1. Table

[3]Available at https://github.com/annalenalamprecht/CoTaPP/blob/main/Lecture%20Notes/06%20Functions%20and%20Modules.ipynb

5 provides an initial structure outlining the goals, prompts, and activities that the teacher can do for both the understanding and applying outcomes of the concept. Finally, mapping those scaffolding techniques to our preliminary framework—abstraction skills, approaches, and stages, as detailed in Section 4.5—yields a new strategy for teaching the concept, as shown in Table 6.

By applying the proposed framework, we can use the original examples (Listings 1, 2, 3, 4, and 5) and exercises (Figures 2 and 3) alongside various techniques (prompts, activities, etc.) to guide and support students in first understanding the concept, and then applying it.

```
def pow(x,n):
    if n == 0:
        return 1
    else:
        return x * pow(x,n-1)

x_to_the_power_of_n = pow(x,n)
```

Listing 1: Recursive implementation to compute xn in Python.**

```
pow(3,5)
    pow(3,4)
        pow(3,3)
            pow(3,2)
                pow(3,1)
                    pow(3,0)
                    return 1
                return 3
            return 9
        return 27
    return 81
return 24
```

Listing 2: Programming run of `pow(3,5)`.

```
def factorial(n):
    if n == 1:
        return 1
    else:
        return n * factorial(n-1)

factorial_of_n = factorial(n)
```

Listing 3: Recursive implementation to compute the factorial of a number, with a similar structure to the `pow` function.

5 Results: Contexts and Instructional Tools

This section introduces a number of learning contexts and demonstrates how the preceding framework (Table 4) can be applied to these contexts.

Write a function to determine the sum
of the first n natural numbers

sum(n) = 1 + 2 + 3 +.....+ n

Figure 2: Problem statement to apply *recursion.*

Base case: n == 1

Recursive case: n + sum(n-1)

Figure 3: Base and recursive case for the problem in Figure 3.

```
def sum(n):
    if n == 1:
        return 1
    else:
        return n + sum(n)
```

Listing 4: Incorrect recursive implementation for the problem presented in Figure 2. A student is not reaching the base case.

```
def sum(n):
    # Base case
    if n == 1:
        return 1
    # Recursive case
    else:
        return n + sum(n-1)
```

Listing 5: Correct recursive implementation for the problem presented in Figure 2.

Each of these contexts is introduced with a description of concepts taught, challenges, and a sample of the work assigned to students. The intellectual challenges are then modeled in the context of our framework, with additional references to relevant specific concepts of abstraction learning from the literature described above. The contexts cover the breadth of undergraduate computer science coursework, from pointers and dynamic memory in an introductory course to an elective course on algorithmic game theory taken by undergraduates in their final year. These contexts were instrumental in building the model and all come from materials taught by various authors of this paper.

We note that the contexts cover a wide range of topics from the computer science curriculum and more than one educational level. There are also stylistic differences between the contexts and variance in their level of detail. We consider this a strength of the paper, as the needs of the educators who will use our framework are different; some may seek more comprehensive guidance on how to use it, while others are only seeking broad directions.

This section provides multiple real-world examples of how the framework can be applied to a context, and how applying the framework can help improve course materials. The framework should

Table 5: Starting point to apply our framework to teaching the concept of *recursion*. Stages refer to the initial version of the framework, as defined in Section 4.5.

Stage	Goal	Prompt	What the teachers can do	Example
U1	Observe that the function (1) has two parts, a base case and a recursive case, and (2) performs operations that are repeated, because it calls itself	Observe the recursive implementation of the pow function	Discuss what is the purpose of the conditional?	Listing 1
U2	Induce/deduce the function is broken down into smaller sub-problems by going into each function until the base case is reached	Trace through the code a specific run of the function	Show an example of manual run of the function, showing individual states; compare to another implementation	Listing 2
U3	Hypothesize the implementation of the function may ensure that all computations contribute to the end result or may be more efficient for some problems; being able to write a recursion when presented with a problem that can be expressed as a recurrence problem	Write a recursive function from a recurrence	Provide more scaffolding to get to the solution, such as showing the recursive case only; showing another implementation of the same problem; showing a similar example	Listing 3
A1	Observe the characteristics of the problem and identify that it can be solved using recursion	Observe the problem definition	Ask the students to recall the characteristics of recursive functions	Figure 2
A2	Induce/deduce the base case(s) and the recursive case(s). Deduce that is better to start from N than from 1	Write the base case and the recursive case	Check that the base case and the recursive case are correct before moving to the implementation	Figure 3
A3	Hypothesize that the errors are preventing the computations to contribute to the end results and the base case to be reached. Or that the working solution contribute to the end results and the base case to be reached	Write the implementation of the solution, test it and adapt it. Do this as many times as necessary until reaching the correct solution	Ask the students what part of the code is working and which one not. Encourage a "refinement" loop till the solution is reach.	Listings 4 & 5

Table 6: Our pedagogical framework, as defined in Section 4.5, applied to the understanding and application of *recursion*.

Stage	Identifying	Moving	Creating	Outcome
U1	Programming constructs: function, if-statement, call to a function. Prerequisite: Students are able to go through the code line by line			Understand the general concept of recursion
U2 (activity 1)	Programming constructs: call to a function (self-reference)	Move between the (0) recurrence, (1) the code implementation, and (2) the trace. Prerequisite: students are able to trace an input		
U2 (activity 2)	Identify similarities between different implementations; similar examples	Move between the recurrence (e.g., x! = x * (x-1)!) and the implementation	Giving a recurrence (e.g., x! = x * (x-1)!), create the implementation	
U3		Move between then recurrence (e.g., x! = x * (x-1)!) and the implementation	Giving a prompt in natural language, create the implementation	
A1	Identify that they could use a recursive function			Implementing a correct solution using recursion
A2		Move from the natural language description, to a representation that is closer to the code implementation		
A3	Error identification (loop) Prompt: what part of the code is working and which one is not?	Moving to the implementation (not perfect) (loop)	Error correction (loop)	

not be seen as a table to complete, but rather a tool to categorize existing instructional material. The examples conclude with the opportunities for improvement that were identified through this categorization.

Insights from the process of applying the framework to the contexts can be found in Section 6 and the refined version of the framework is presented in Section 7. In the contexts the abbreviations U1, U2, U3 and A1, A2, A3 are used to refer to the stages in Table 4.

5.1 Context 1 - Concept of Memory and Pointers in C for CS1

5.1.1 Context of the Course. It is a one-semester, 11-week CS1 module taught in C in a research intensive university in the UK. Each week, students receive 1 hour of recorded lecture, 1 hour of live coding on solving problems or demonstrating the development environment, and 3 hours of laboratory session to support assessed and non-assessed exercises. The module covers basic data structures like linked lists, stacks, queues, and arrays, and algorithms such as searching, sorting, recursion. There are approximately 340 students, with some have completed *Advance level (A-level)* mathematics, and computer science. A level qualifications are subject-based qualifications that can lead to university, further study, training, or work in England, Wales and Northern Island.

Before the material covered in the context below, students will have learned basic C programming principles, including primitive variables, conditionals, loops, functions, and static arrays. This unit introduces a large number of new concepts, some of which prove challenging for students, including data representation, pointers, the address-of and dereference operators, and C-style strings.

The learning focus for this unit has been reconsidered using the our framework to help students understand and apply moving between different levels of abstraction of data representation. The students are exposed to aspects of both Stage 1 (Observe) and Stage 2 (Induction and Deduction) for both Understanding and Application, with a focus on the Moving abstraction skill.

The current version of the unit moves through this material rapidly, causing many students to struggle, especially with concepts not addressed directly or sufficiently. As a result, we have used the abstraction framework to identify opportunities to introduce what we hope to be small but effective interventions for the programming concepts in this unit.

5.1.2 Understanding Data Representation. CS1 students learning C must understand machine-level representation of data before they are able to properly manipulate vector data. First, students learn that all data are represented in binary form, so that two different pieces of data belonging to two different types can have identical representation. We apply the stages and understanding from our pedagogical framework.

Stage U1: Using base conversion between binary and decimal and using the ASCII table, students see that both an `int` value of 65 and that a `char` value of 'A' are stored as `01000001` in binary, or `0x6B` in hexadecimal.

```
1 int a = 65;
2 char b = 'A';
```

0x00000400	0	0
0x00000401	0	0
0x00000402	0	8
0x00000403	A	4
...	x	x
0x000008A4	0	0
0x000008A5	0	0
0x000008A6	0	0
0x000008A7	0	0
0x000008A8	0	0
0x000008A9	0	0
0x000008B0	0	0
0x000008B1	0	0

Figure 4: Pointer representation in sequential memory

Stage U2: Using deduction, students infer that printing 65 as an `int` will print 65 in line 3, while printing the same bit-pattern as a `char` in line 4 prints "A".

```
1 a = 'A';
2 b = 65;
3 printf("%d\n", a);
4 printf("%c\n", b);
```

Stage A2: Students are expected to deduce that a hexadecimal value in a `char` context will print as the corresponding ASCII character representation and that printing a `char` value in int context will print the corresponding integer ASCII index.

```
1 printf("%c\n", 0x6B);
2 printf("%d\n", 'k');
```

5.1.3 Moving Between Abstractions of Pointers. The unit also teaches students about pointers, `int` arrays, and the address-of and dereference operators. Students are not expected to achieve a complete understanding of pointers and memory models in this unit. The goal of the unit is to move students through the first two stages (observation and induction, then induction and deduction) for both understanding and applying. In both cases, we use moving as the abstraction skill to leverage.

We have these specific outcomes for this part of the unit:

- to correctly assign referents to pointers using the address-of operator
- to correctly access and/or alter the value of the referent of a pointer using the dereference operator

Here students observe the relation between declaring pointers, assigning values to pointers, de-referencing pointers, and assigning value to pointer referents. Here they also observe the dynamic aspects of the pointers.

The current unit material emphasizes a notional machine representation of sequential memory addresses, and shows that one address can contain a value that is the address of another byte, as seen in Figure 4.

Unfortunately, Figure 4 is not sufficient for students to understand the operation of pointers. We introduce an additional notional machine to the instruction, bridging from the existing memory diagram to block-and-arrow memory diagrams showing the pointer-to-addressee relationship. This is as a result of deficiencies identified as

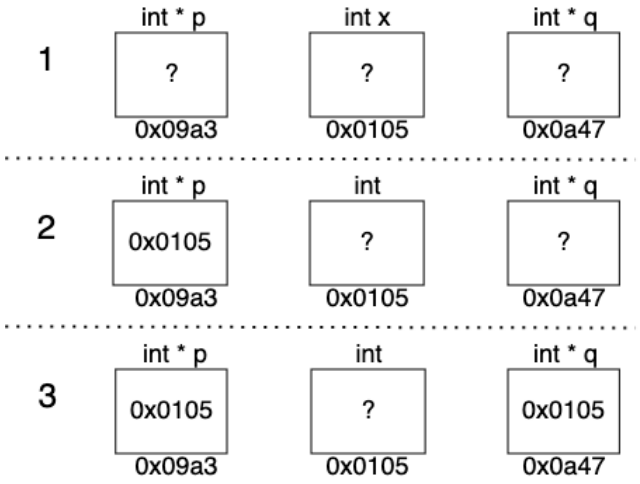


Figure 5: Pointer representation without sequential memory

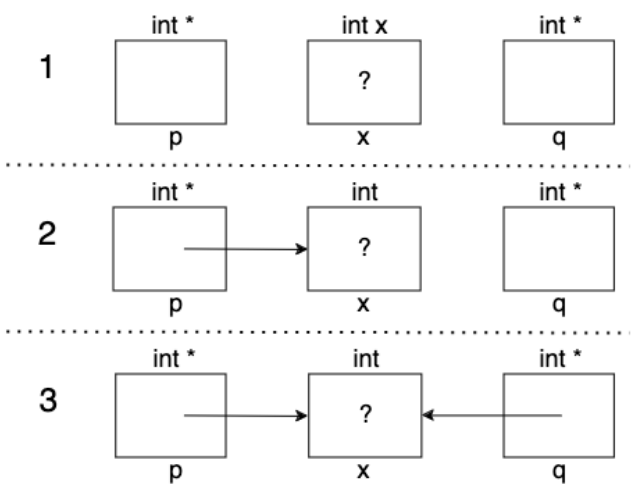


Figure 6: Pointer representation with referencing arrows

a result of reflecting on this context with our proposed framework. We will begin by presenting the following code to students:

```
int x;
int *p, *q;          // 1
p = &x;              // 2
q = p;               // 3
printf("%p\n", p);  // all three lines print
printf("%p\n", q);  // the same hex value
printf("%p\n", &x); // the address of x
```

The second notional machine shows the relationship between one integer primitive and the pointers, but removes the concept of sequential memory locations, simply showing that a sequence of 8 bytes[4] (a pointer) can contain the address of another byte. This is seen in Figure 5. Our framework enabled us to identify the scaffolding notional machine to understand the intricacies of pointers.

This will be used as a bridge to block-and-arrow memory diagrams. In these diagrams, pointers don't have values (e.g., a hexademical value); they have arrows that point to other bytes. This is a move from one level of abstraction (8 bytes containing the address of another byte) to the meaning of that concept, which is that one sequence of 8 bytes points to another sequence of bytes. The second version of the diagram (seen in Figure 6) represents the same code, but with this higher-level abstraction.

5.1.4 Using the Dereference Operator. The student will then be given the following code and associated memory diagram showing states 1–3, seen in Figure 7:

```
int x = 42;
int *p;              // State 1
p = &x;              // State 2
printf("%d\n", *p);
```

[4]The number of bytes used to represent a pointer in C can differ based on platform. However in this example we will assume 8 bytes are used to store a pointer, which is common on 64-bit systems.

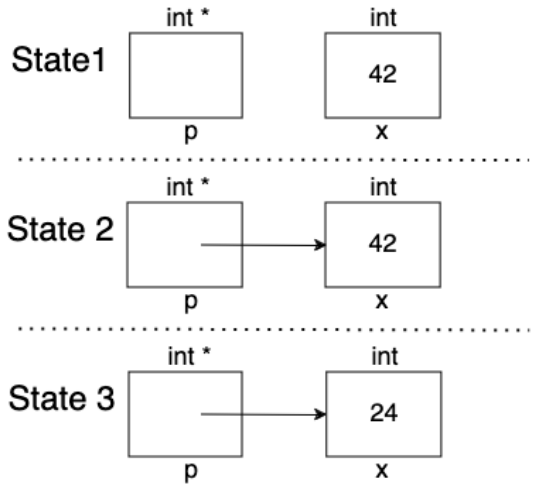


Figure 7: Example memory model showing first three states

```
*p = 24;             // State 3
printf("%d\n", *p);
printf("%d\n", x);
```

The student will then asked to modify the memory diagram to show the result of the following line of code. The student should produce the diagram shown as State 4 in Figure 8.

```
*p = 31;             // State 4
```

At the end of the unit, students will be expected to understand all of these concepts and how C represents both integer and character data and how those two concepts are simply different interpretations of the same basic underlying data.

5.1.5 Pointers to Arrays of Integers. Now pointers are used to reference statically-allocated arrays. Students are assumed to already have an understanding of declaring and using arrays.

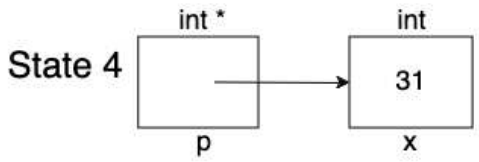


Figure 8: Student-produced memory model showing the final state

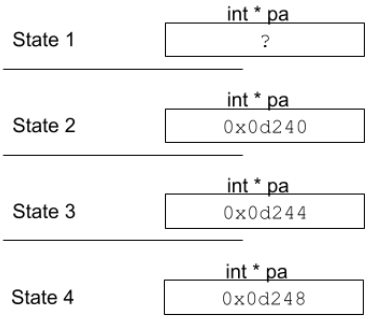
The students are introduced to the code below, along with the model in Figure 9. That model traces the changing value of pa to reference sequential elements of a. Both the index and dereference operators are used to print both the addresses and contents of the array using both variables. In each state, students will see that the addresses and values are identical.

```
1  int a[5] = {1,2,3,4,5};
2  int *pa;                  // State 1
3  pa = a;                   // State 2 - index 0
4  printf("%d\n", a[0]);
5  printf("%p\n", pa);
6  printf("%p\n", &a);
7  printf("%d\n", a[1]);     // State 3 - index 1
8  printf("%d\n", *(++pa));
9  printf("%p\n", ++pa);     // State 4 - index 2
10 printf("%p\n", &(a[2]));
```

A1 Identify and Creating: The goal is to understand the relationship between pointers, arrays, array indexes, and the index and dereference operators ([] and *, respectively). The above code is presented to students along with the corresponding memory model. One of the first things to observe is the access of elements through two mechanisms. One is the indexing and another is through the pointer. At this stage students are encouraged to understand a memory representation of array a and pointer pa. They should be able to complete this code to traverse the array.

A2 Identify and Creating: In this stage, the students will deduce the value of x using the listing below. They may take the help of the memory diagram above. Given that students are already familiar with iterating using loops and arrays, students will be asked to create 1) a loop to traverse the array through both pointer and index, and 2) a loop to traverse the array showing the address location of using both pointers and array index.

```
1 int a[5]={1,2,3,4,5};
2 int *pa, x;
3 pa=a;
4 x=*(pa+2);
5 x=*pa+2;
6 x = *pa++;
7 x = (*pa)++;
8 x = *--pa;
9 pa+=3;
```



Address				Value
d	2	4	0	1
d	2	4	4	2
d	2	4	8	3
d	2	4	c	4
d	2	5	0	5

Figure 9: Model of an array and a pointer used to reference multiple indexes

```
10 x=*pa;
11 x=*(pa-4);
```

5.1.6 Conclusion. In summary, this context introduces the concepts of memory and pointers in C, guided by the proposed framework. The progression is as follows:

(1) Understanding the data representation by observing that different data type may have the same underlying bit-pattern (Section 5.1.2).
(2) Moving between bit-representation and conceptual representations of pointers (Section 5.1.3). This is achieved by guiding the students from a linear representation (Fig. 4) of memory, towards a more schematic representation (Fig. 5), that eventually removes the addresses altogether (Fig. 6).
(3) Using the dereference operator to access memory through pointers (Section 5.1.4). This builds upon the schematic representation from the previous step, and illustrates how modifications of the underlying data affect the representation (Figs. 7 and 8).
(4) Discussing pointers to arrays of integers (Section 5.1.5), where students are shown the fact that pointers can be modified, much like other integer variables, in order to access data located sequentially in memory (Fig. 9).

5.2 Context 2 - Algorithmic Game Theory

5.2.1 Context of the Field. Algorithmic game theory is an interdisciplinary field that combines elements of economics, mathematics, and computer science to study strategic interactions among rational and intelligent decision-makers. It extends traditional game theory by focusing on the computational aspects of game-theoretic problems, addressing questions about how strategies can be computed efficiently and how strategic behavior can be algorithmically analyzed. The field explores the design and analysis of algorithms within strategic environments, aiming to understand, among others, the complexity of finding equilibria and the efficiency of various outcomes in the presence of strategic behavior.

While not universally included in every undergraduate Computer Science curriculum, Algorithmic Game Theory is more common in advanced courses, especially in programs emphasizing theoretical computer science. Typically offered as an elective or as part

of a broader course on algorithms, optimization, or computational economics, Algorithmic Game Theory remains somewhat niche but but has clear relevance in areas like online platforms and auction design.

Dixit [25] claims that too many teachers and textbooks treat the subject in a very abstract and formal way, thus losing its advantage of applicability to numerous real-life decision-making scenarios, and instead suggests methods for teaching game theory using interactive games to be played in the classroom. Holt and Capra [42] also use classroom games to illustrate real-life applications of the prisoner's dilemma and stimulate discussion of a wide range of topics in economics and game theory. In this case study, we will discuss how one can use real-life examples in the classroom as a stepping stone to abstracting complex game theoretic notions and concepts, via the lens of three different systematizations, all while applying our framework.

5.2.2 Context of the Module. This case study explores a Year 3 module taught predominantly to Computer Science students (though it is open to students who may satisfy the prerequisites, e.g. Natural Science students) in a research-intensive University in the North East of the United Kingdom. It is delivered across 10 teaching weeks, with 2 hours of lectures per week and no practical classes, though formative exercises are handed out each week and students are expected to work on those. The module's title is Algorithmic Game Theory; the topics it considers are strategic games and Nash equilibria, with an emphasis on bimatrix games as a means of introduction, extensive games with perfect information, mathematical and algorithmic foundations of market equilibria, routing games on networks, congestion games, and combinatorial auctions. The module, by its very nature, requires students to think critically about strategic interactions and model complex scenarios using mathematical tools. This necessitates a pedagogical strategy that emphasizes the development of abstraction skills, allowing students to distill real-world problems into manageable and analyzable models.

5.2.3 Application of our Framework Within this Context. Each teaching session is designed not only to impart knowledge but also to engage students in the process of abstraction, which is paramount in studying more and more complex game theoretic scenarios as we move on to more advanced topics. For instance, when introducing bimatrix games and Nash equilibria, students are guided through the transition from real-life scenarios to mathematical representations. This involves outlining a game theoretic problem as a problem one would face in real-life, identifying the players, strategies, and payoffs, and then formalizing these elements into a structured game model. By consistently linking theoretical content with practical examples, students learn to see the relevance and applicability of abstract models in the context of the module. Linking this to our proposed framework, and the three inferential stages, through exposure and engagement in practical examples, students first observe concrete things (Stage 1); through analysis — still on the concrete example rather than on a generic / abstract concept — students make inferences about the specific observations made (Stage 2); finally, students use the observations made for the concrete example to first think hypothetically and then argue more generally about the abstract concept (Stage 3).

In the following, we will look at specific abstract concepts taught within the Algorithmic Game Theory module and how we apply our framework to introduce those concepts to students and help them to achieve a good understanding of them while, in the process, developing their abstraction skills. The concepts that we will look at are:

(1) *Second-price sealed bid auctions*; there is a single item for sale and a number of bidders. Each bidder submits a sealed bid to the seller. The highest bidder obtains the item and pays the second-highest bid for it. A bidder's valuation is the maximum amount they are willing to pay for the item. The payoff of a player if she does not obtain the item is 0; her payoff if she obtains the item is the difference between her valuation and the price she pays for it (second-highest bid). It can be shown that truthful bidding, i.e., submitting one's true valuation for the item, is a *dominant* strategy in Second-price sealed bid auctions, that is a strategy that is better than any other strategy for the player, no matter how that player's opponents play.
(2) *(Pure) Nash equilibrium*; in the early introduction to the module, students learn about the simple concept of a (pure) Nash equilibrium. In a strategic game, each player has a set of actions from which they pick one to 'play'; she also has a payoff function which maps every action profile (namely, combination of actions, one per player) to a real number that, in a way, represents the 'happiness' of the player in that given action profile. A standard assumption is that players make their decisions on which action to choose simultaneously and once and for all, i.e. they may not change their minds. A (pure) Nash Equilibrium is a steady-state in the game, namely it is an action profile where, given other player's strategies / actions, no player wishes to unilaterally change their own strategy / action.
(3) *Extensive games with perfect information*; these games model strategic interaction situations with a sequential nature, in which each player is free to change her mind as events unfold and does not need to stick to a plan of action from the start of the game; each player is also informed fully about other players' actions prior to when they need to take an action. Games like chess or tic-tac-toe are classic real-life examples of such games.

We note that the above paradigms (an auction; a Nash equilibrium; an extensive game) are all abstract concepts but there exist concrete real-life examples that a teacher can point to, which students can relate to (eBay; everyday interactions where there might be differing interests; chess, respectively). Therefore, it is unproblematic to find very specific examples that students can relate to in order to motivate an initial analysis of properties of these concepts, thus leading to a higher level understanding of the concepts themselves as well as their properties; recall that the first stage for understanding a concept in our proposed framework (i.e., stage U1 for the identifying skill) involves making observations, and presenting students with relatable examples can support this process.

In the following subsections, we present an in-depth analysis of the application of our framework in the first paradigm above, namely to the teaching of second-price sealed bid auctions. We also

highlight ideas and elements from the application of pre-existing systematizations to the above paradigms that gave us initial insights into the important aspects of the development of abstraction in the context of this module. These insights helped form aspects of our framework in turn; indeed, recall that the development of our framework was an iterative process of theoretical analysis and reflective practice across multiple teaching contexts, including this one.

5.2.4 Auctions Paradigm. To illustrate the application of our proposed framework in detail, we will first look at how a model of concept formation, specifically the APOS model (Actions, Processes, Objects, and Schemas)[5] [17], is used to help students form an understanding of the concept of Combinatorial Auctions. We will then show how our framework builds upon, e.g. the steps of the APOS model, to enhance the development of several of the identified key abstraction skills.

Let us consider the following setting: students have been introduced to the concept of an auction in general and have been presented with the basic definition of a second-price sealed-bid auction as a process where bidders submit simultaneous sealed bids to the seller of an item and the highest bidder wins the item and pays the value of the second-highest bid. Suppose further that the students know / have been taught that a second-price sealed-bid auction can be formulated as a strategic game where the players' payoffs are clearly defined for a given bidding profile, i.e., combination of bids, one per player / bidder; a player's payoff is 0 if she does not obtain the item in the given bidding profile, and is equal to her (valuation - price paid) if she obtains the item. The APOS model can be applied as follows:

- Students participate in a mock second-price sealed-bid auction where they bid for a box of chocolates. (*Actions*)
- We then collectively discuss the process of determining the winning bids and payments. We ask questions and prompt students to explain their process of deciding what bid to place, as well as how changing their bid may change the outcome of the auction for them. (*Processes*)
- Students are then encouraged to view the auction mechanism itself as an object that can be analyzed and optimized. Here, students must make inferences about the observation of possible outcomes and attempt to generalize what these mean for an arbitrary player's dominant strategy in this type of auction. (*Objects*)
- Finally, the students are able to build a schema around different types of auctions (1st price, 2nd price, all-pay, etc.) and their properties, e.g., dominant strategies or Nash equilibria. (*Schemas*)

We now further enhance the above process through the application of our framework to highlight the development of individual abstraction skills. This insight allows the teacher to adapt the teaching, to ensure a suitable progression in each skill and as a whole. We present the enhanced process of teaching students about second-price sealed-bid auctions in detailed steps below. Something that should come out of this process is that students understand that *bidding truthfully* (i.e., bidding their true valuation for the item) is the best action regardless of other players' bids in a second-price sealed-bid auction. Within each step, students are given prompts or are expected to reach certain subgoals of understanding or abstraction; where these are explicitly discussed, we discuss them also in the context of the three inferential stages proposed in our framework:

Step 1. Prior to engaging the students in the mock auction for the box of chocolates, we bring in the classroom two people who have been briefed about the 'experiment' (of developing students' abstraction skills using the APOS model as above), e.g. two PDRAs, let us call them Bob and Alice. The PDRAs participate in an auction for an apple to illustrate to the students the bidding process: Bob announces to students that his valuation for the apple is £1, while Alice announces that her valuation is £2. They both submit their bids to the teacher who proceeds to open them and announces to the students that Bob submitted a bid of £10, while Alice submitted a bid of £2. Bob is declared the winner of the auction and needs to pay the teacher £2, i.e., the second highest bid. The teacher asks students: "What is Bob's payoff for obtaining the apple?" (This is a Stage 1 prompt based on the observation of the 2-player auction, where players' valuations are known.) Bob's payoff is negative, namely -1. (This is a Stage 1 subgoal for the students to **identify** what the payoffs are based on their observation of the concrete 2-player auction between Bob and Alice. At the same time, here students should be able to **apply** their prior understanding of payoffs, as they have developed it in the context of strategic games, to this new concept of auctions, so this also serves as a Stage 3 subgoal of **applying** some prerequisite knowledge.). If students are not able to reach this subgoal, there are further prompts that can be asked; e.g. "Let us recall: how are payoffs calculated in an auction?" (Stage 1 prompt for the skill of **identifying**), "What are the specific values that need to be input in the calculation in this concrete example?" (Stage 2 prompt for **identifying**) etc. In this step, not only students observe the bidding process but are developing the skill of **identifying** that this specific situation can be modeled with the notion of an 'auction' as they have preliminary understood it via the initial discussions on auctions that occurred already before this process.

Step 2. The teacher now poses the question "What would have happened if, instead of £1 and £2, Bob and Alice's valuations were £11 and £0.5, respectively, while bids remain the same?" (This is a Stage 2 prompt where students are encouraged to make deductive inferences about their observation of the process so far.) The answer is that Bob would have still won with the bid of £10 but his payoff would now be positive, namely 0.5. Students should now reach the Stage 2 subgoal of being able to deduce the complexities of an auction in that they **understand** that a player's payoff is dependent on their valuation and their bid, as well as the other players' bids. E.g., Bob should never bid more or less than £1: if he bids more he risks Alice bidding higher than £1 but lower than Bob, thus he risks ending up with negative payoff; if he bids less than £1 then he misses a potential opportunity of obtaining

[5]The APOS model focuses on the mental construction of mathematical concepts through the four stages of Actions, Processes, Objects, and Schemas.

the apple and a positive payoff if Alice bids less than £1. In this step, the teacher is merely asking questions for students to reflect on what strategy / bid might be appropriate but without prompting any further analysis. This step can aid in the **understanding** of an auction more broadly as an object, though at a very preliminary stage and possibly only combined with the subsequent steps below. As such, there is no expectation that students reach the Stage 3 subgoal of understanding that truthful bidding is dominant regardless of other players' bids just yet. Some students may already reach that subgoal in this step of the process, though it is expected that most will require going through the subsequent steps.

Step 3. The students are now asked to participate in the mock auction for the box of chocolates as described above in the application of the APOS model. Now, students can **apply** their understanding (as abstract as it has been formed so far) about second-price sealed-bid auctions to a more complex example, where in particular they are not aware of others' valuations. In this step, a student that has already managed to **understand** that they are better off bidding their true valuation for the item no matter what others bid has also been able to **move** between levels of abstraction. Specifically, the student has moved from the very concrete example of two players with known valuations to the more abstract level of understanding that truthful bidding is dominant regardless of number of players, valuations, and others' bids. For those that have not yet managed to understand that, Step 4 offers more prompts.

Step 4. The teacher follows up with more prompting questions after all students' bids have been opened. Let us consider an example where there are four students, A, B, C, and D, with bids £10,000, £500, £10, £1, respectively. The winner is A who has to pay £500 for the box of chocolates and certificate. The teacher asks A what her valuation is for the item(s) and A replies that it is only £5. The teacher asks "Why did you choose to bid £10,000?" and A replies "I thought that nobody else would bid this much and I wanted to win." The teacher then reminds A that obtaining the object alone is not the objective of the auction — they must consider their payoff in doing so; A's payoff is now -495. The teacher then prompts A to think what she would bid if she could bid again. (This is a Stage 3 prompt for **identifying** as well as the outcome skill of **understanding** where students are required to think hypothetically using their observations.) At this point, A (resp. the other students) may have already been able to **identify** truthful bidding as the best strategy (Stage 3 subgoal). If not, the teacher walks A through different scenarios of possible bids she may place, as well as possible bids that the other players may place.

5.2.5 Nash Equilibrium Paradigm. In this section, similarly to the previous one, we present how Concreteness Fading (CF) [28] relates to our proposed framework by applying them both to the concept of a Nash equilibrium. CF involves starting with concrete examples and gradually moving towards more abstract representations. It is worth remembering that the emphasis of a framework of pedagogic practice is placed on what *the teacher* does, while models of abstraction focus on what *the students* do. With that in mind, let us explore the relation between CF and our proposed framework in the teaching of the concept of a Nash equilibrium.

Step 1. We begin a discussion with students through looking at the simple, real-world game of Rock-Paper-Scissors. Students play in pairs and record outcomes.

This activity corresponds to the first step in CF, *Concrete*. In our framework, this corresponds to inferential Stage 1, where students make observations about the concept.

Step 2. We proceed by instructing students to work in larger groups: in their groups, students transition into representing the Rock-Paper-Scissors game with a payoff matrix, explaining and understanding how each outcome corresponds to different payoffs.

This corresponds to the second step in CF, *Less Concrete*. This step encourages students to continue to Stage 2 in our proposed framework, since it encourages discussions that lead to both deductive and inductive inferences about the concept.

Step 3. We finally introduce the concept of Nash equilibrium and discuss how it can be found in the payoff matrix.

This final step corresponds to the third and final step in CF, *Abstract*. This is where instructors help students generalize their observations from the previous two steps into the Nash equilibrium, thereby being able to reason hypothetically about other, similar situations.

5.2.6 Extensive Games Paradigm. For the final concept, similar to the previous two, we present how Sfards's Model [75] on Interiorization, Condensation, and Reification can be applied to the teaching of extensive games with perfect information, and how it relates to our proposed framework.

Step 1. Students are instructed to practice playing an easy extensive game with perfect information like Tic-Tac-Toe in pairs. In that process, students are encouraged to think ahead about the possible outcomes of the game and how different actions they take each time it is their turn to play (and every subsequent time) may influence their payoff.

This step corresponds to *Interiorization* in Sfard's framework and inferential Stage 1 in our framework, as it encourages students to observe the phenomenon and to make inductive inferences about it. Additionally, the latter part of this step (thinking about outcomes) encourages students to move towards Stage 2 of our framework.

Step 2. We then guide students to summarize their strategies into decision trees / game trees where they need to consider and show the possible moves and outcomes.

This step corresponds to *Condensation* in Sfard's framework and inferential Stage 2 in our framework, as it encourages students to continue the inferences they made in Step 1 to include both inductive and deductive reasoning.

Step 3. Finally, students see the entire game tree as an abstract object, allowing them to reason about strategies and outcomes at a higher level.

This corresponds to *Reification* in Sfard's framework, and inferential Stage 3 in our framework, as it allows students to generalize their observations from the previous steps into its own concept (the game tree, in this case), and later apply it in other situations. For example, a direct consequence of reaching this stage is understanding the backward induction process which when performed on a game tree can identify what is known as Subgame Perfect Equilibria.

5.2.7 Discussion. By applying our framework as well as the above models in these paradigms, students can progressively develop a deeper and more structured understanding of the algorithmic game theory topics we look at, transitioning from concrete experiences to sophisticated theoretical insights. It is also clear that our framework has links to existing systematizations; indeed, the outline of the application of the above systematizations within the context of the Algorithmic Game Theory module was a part of the iterative process used for the design of the framework. Identifying shared elements and areas of divergence informed the adaptation and integration of key ideas from these systematizations towards the formation of the final version of our framework. For example, as we saw in Sections 5.2.5 and 5.2.6, both Concreteness Fading and Sfard's framework have a similar progression when focusing on what *students* need to do. There are, however, minor differences in the extent of the first step. In CF, the first step only involves making observations, while it also involves making basic inductive inferences in Sfard's framework.

5.3 Context 3 - Concurrent Programming

Concurrent programming is an important concept since it allows utilizing the full computing power of modern multi-core CPUs, and it is therefore often included in university-level CS programmes [45]. It is a topic that is often perceived as difficult by students, likely because students are no longer able to utilize trial-and-error to explore the environment and have to shift towards a more formal approach instead [47, 81]. This is in large part because the concepts used in concurrent programming differ from concepts that students are already familiar with in that the new concepts leave much behavior undefined. As such, observing a running program's behavior does not necessarily allow concluding that the observed behavior is well-defined.

Concurrent programming requires students to take a step back and re-consider their approach to programming (as argued by Kolikant [47]), it is a topic that is typically taught later in a CS education. Therefore, it demonstrates the application of the framework to more advanced topics in addition to the foundational topics discussed in many of the other case studies in this section. As such, it covers all of the abstraction skills outlined in Section 4.2.2.

5.3.1 Context of the Course. The course examined in this section is given towards the end of the second year of a three year long Bachelor's programme in Computer Science at a large university in Sweden. The course runs for a period of 10 weeks. Its main focus is a set of computer lab assignments where students implement functionality in the educational operating system Pintos.[6] The course also includes 4 lectures that introduce concurrency (the theory on operating systems has been covered in a previous course), two lectures that introduce concepts related to Pintos, and ends with a final exam where students are asked to synchronize simple programs[7].

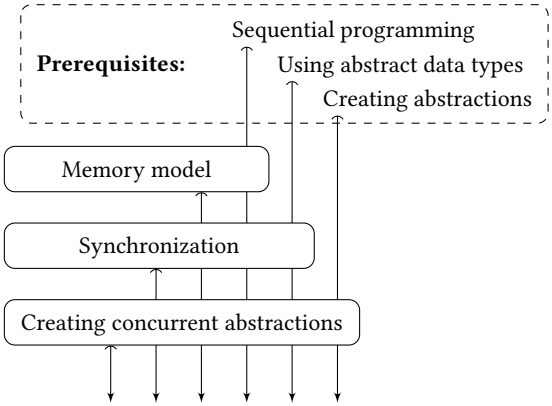


Figure 10: Overview of the expected prerequisites (in the dashed box), the concepts covered (solid boxes), and the dependencies between them (arrows).

The remainder of this context is split into two parts. The first presents the contents of the course in general and its relation to the framework. The second part briefly covers practical considerations.

5.3.2 Concepts in the Course. The course assumes that students are already familiar with sequential programming. As such, the course starts from there and covers the differences between sequential and concurrent programming. At a high level, the differences between sequential programming and concurrent programming in the shared memory model can be summarized into three high-level concepts that students need to learn:

(1) The *memory model* that describes the behavior of program execution,
(2) *Synchronization* using *synchronization primitives*, and
(3) *Creating abstractions* that are suitable to concurrent systems.

As shown in Figure 10, these concepts cover concurrent programming by extending the computational model used for sequential programs. Therefore, students are assumed to be already familiar with sequential programming. In particular, they are expected to be able to trace and write programs that involve basic control structures, functions, custom data types, pointers/references, and the ability to create data- and procedural abstractions. Step (1) above covers the memory model in concurrent programming by illustrating the problems that are unique to concurrent programming, namely that data shared between multiple threads need to be protected. It is thereby mainly focused on the *identifying* abstraction skill, as students need to identify shared data and problematic code that uses this data. Step (2) builds upon the insight from step 1 and covers how shared data can be protected, which involves the *move* skill, as students need to *move* between abstraction levels to eventually come up with general rules to synchronize programs. Finally, step (3) combines both of them and explores how the memory model and the need for synchronization impacts the design and

[6] https://www.scs.stanford.edu/10wi-cs140/pintos/pintos.html

[7] This course is described in further detail in [82].

implementation of abstractions for use in a concurrent system. As such, it covers the *creation* of abstractions in concurrent systems.

5.3.3 Course Content: The Memory Model. The fundamental idea behind concurrent programming (in the shared memory model) is to let each program (*process*) contain multiple *threads*. Each thread executes instructions sequentially, exactly like in sequential programs. All threads execute concurrently and are thus allowed to execute on different CPU cores, which in turn lets the program utilize the multiple CPU cores that are available in most modern systems. Since different systems have different number of CPU cores and different number of threads running only weak guarantees for thread execution order are provided. It is typically only possible to assume that all threads will eventually get the chance to execute, not that they execute at the same time (i.e., in parallel), or even at the same speed relative to each other.

Different threads communicate through the shared memory. As such, the semantics of memory accesses is central to concurrent programming in the shared memory model. In sequential programs, memory accesses appear to occur in the order they appear in the source code. In practice, however, both compilers and hardware reorder memory accesses to improve performance. The modifications are done in such a way that they are not observable in sequential programs. They are, however, visible in concurrent programs. Together with the weak guarantees for thread execution, this means that students need a good grasp of the semantics of the programming language in order to properly reason about the behavior of programs. This is something that students often struggle with [80, 81].

```
1  int shared = 0;
2
3  void fn(void) {
4    for (int i = 0; i < 100000; i++)
5      shared += 2;
6  }
7
8  int main(void) {
9    thread_new(&fn);
10   printf("shared␣=␣%d\n", shared);
11   return 0;
12 }
```

Listing 6: Example of a program that illustrates the weak guarantees of thread execution. Depending on the implementation of `thread_new`, a small delay may need to be inserted before line 10.

The first stage for understanding a concept in our proposed framework (i.e., stage U1 for the *identifying* skill) involves making *observations*. Since students are expected to be familiar with sequential programming in a C-like language, a suitable starting point is the program in Listing 6.[8] The `main` function starts a new thread that executes `fn` by calling `thread_new`,[9] and then prints the value of `shared`. The started thread adds 2 to `shared` 100,000 times. When the program is run (either by the teacher or by the students themselves), it prints a number between 0 and 200,000, and usually a different number each time. This is likely contrary to students' initial predictions, and illustrates the non-deterministic nature of concurrent programs.

The teacher uses the illustration of non-determinism as a starting point for a discussion about possible solutions to the problem, ideally in a situation where students are able to modify the code and test it during the discussion. This encourages cycles of inductive and deductive reasoning about the semantics of memory accesses, which corresponds to stage U2 of our proposed framework for the *identifying* skill. Eventually, the students are likely to arrive at a solution similar to the one in Listing 7, which adds the variable `done` to let the `main` thread *wait for* `fn` to complete.

```
1  int shared = 0;
2  bool done = false;
3
4  void fn(void) {
5    for (int i = 0; i < 100000; i++)
6      shared += 2;
7    done = true;
8  }
9
10 int main(void) {
11   thread_new(&fn);
12   while (!done)
13     ;
14   printf("shared␣=␣%d\n", shared);
15   return 0;
16 }
```

Listing 7: Example of a program that illustrates the issues with shared data.

The code in Listing 7 usually *appears* to work well, even though it is undefined according to the C memory model [10]. In this particular case, the program would be correct under stronger memory models, such as *sequential consistency* [49] or *total store ordering* [74]. The program is, however, likely to work since compilers are unlikely to utilize the flexibility allowed by the C memory model unless optimizations enabled. As such, the problem can be illustrated by instructing students to enable compiler optimizations. This makes the compiler use assumptions from the memory model and transform the `while` loop into code equivalent to the code in Listing 8. The effects of the transformation is not easy to observe in practice, since the loop on lines 5–6 will be turned into a single assignment. The discussion around it is therefore best led by a teacher. The effects can be observed by inserting a call to `sleep(1)` before line 7, which causes the program to hang indefinitely. To understand why it is, however, necessary to examine the generated code.

This teacher-led discussion is a clear example of stage U2 in the the framework for *identifying* properties of the memory model. Students made a deductive inference about how to solve a particular problem. With the help of a prompt from the teacher (enable

[8] Of course, this program does not need concurrent execution in its current form. It is, however, trivially extensible to two threads that are performing similar computations concurrently in order to utilize multiple CPU cores.
[9] The implementation is available in Appendix A.

```
if (!done) {
  while (true)
    ;
}
```

Listing 8: Transformation of the loop in Listing 7 by the compiler.

optimizations), they received new information that invalidated the previous deduction. As such, they need to revise their understanding of the memory model through inductive reasoning. The goal is to move towards the insight that all shared data needs to be protected. Since the tools needed to protect shared data have not yet been introduced, it is not possible to fix the problem in Listing 7. Instead, the teacher continues the discussion using non-optimized programs to illustrate another aspect of the memory model to help students to move to stage U3 of our framework, which will be required to continue with the next topic.

```
1  int shared = 0;
2  bool done = false;
3
4  void fn(void) {
5    for (int i = 0; i < 100000; i++)
6      shared += 2;
7    done = true;
8  }
9
10 int main(void) {
11   thread_new(&fn);
12   for (int i = 0; i < 100000; i++)
13     shared += 5;
14   while (!done)
15     ;
16   printf("shared = %d\n", shared);
17   return 0;
18 }
```

Listing 9: Example of a program where two threads access the variable `shared` concurrently.

In the next step, the teacher adds a new loop that increments `shared` 100,000 times to the `main` function, as shown in Listing 9. This loop therefore executes concurrently with the loop inside `fn`. The teacher then asks students to reason about the output of the program. At this stage, students are likely to predict that the program prints 700,000 every time. However, running the program reveals that this is not the case. The reason for this, which the teacher likely has to point out to students, is that each increment operation consists of two separate memory accesses. Thus, even if we assume that memory accesses happen in program order, we can realize that this might result in certain additions being lost. For example, if both threads load the current value of `shared` (e.g., 100), add 2 or 5 to their local copy, and store the result back into `shared` at the same time, one of the values will be overwritten. Since this issue requires that threads perform the above operations in parallel, this only happens for some of the additions. This is why the result differ between different program executions.

As we can see, this illustrates another way in which accessing shared data concurrently may cause unexpected behavior. However, they can be unified and verified using the hypothesis that *concurrent access to shared data is undefined*, meaning that *all access to shared data needs to be protected*. This represents stage U3 of the framework for *identifying* the abstraction of the memory model. It is the level students need to be able to properly reason about synchronization. As we shall see, they will practice applying the memory model in parallel with understanding the next topic in the course.

5.3.4 Course Content: Synchronization. After reaching a sufficient understanding of the memory model, the students can continue to learn *synchronization*. Ideally, students have reached stage U3 at this point. However, as we saw above, synchronization is often needed to affect the execution environment, students may only have reached partway through stage U2 when synchronization is introduced, and have to pursue both topics in parallel.

In the shared memory model, *synchronization* involves using *synchronization primitives*[10] to *wait* for other tasks to complete, or to *avoid* concurrent access to shared data (i.e., ensuring *mutual exclusion*). There are many synchronization primitives available, but we start with the *semaphore* to allow *waiting* for another task to finish, as was required in Listing 7.

```
1 struct semaphore;
2 void sema_init(struct semaphore *s, int value);
3 void sema_up(struct semaphore *s);
4 void sema_down(struct semaphore *s);
```

Listing 10: The public interface of a semaphore.

The first stage of learning semaphores (i.e., stage U1 for *identifying* the abstraction) is to learn the semantics. This also doubles as stage A1 for the memory model, since we apply the abstraction of the memory model to reason about other abstractions. First, the teacher covers the formal semantics: a semaphore is a data structure that has two operations after initialization as shown in Listing 10.[11] The actual contents of the data structure is hidden, but it can be considered to have a counter that must remain non-negative. The counter may be initialized to any positive value when the data structure is created, but may only be manipulated through the operations *up* and *down* afterwards. The *up* operation increases the value by one, and *down* decreases it by one. To ensure that the counter remains non-negative, the *down* operation waits if the counter would be negative after it has been decremented, which makes the calling thread wait until another thread calls *up*. Because of this usage, the two operations are sometimes called *signal* and *wait* respectively. To help students visualize how semaphores behave in the context of a program, a visualization tool like Progvis [82] might also be used. Understanding and using semaphores without seeing the implementation in this way is a good example of hypothetico-deductive thinking.

[10] Or *atomic operations*, but we exclude them as they are not always covered in detail in introductory courses on concurrency.
[11] The implementation is available in Appendix A for reference.

```c
int shared = 0;
struct semaphore done;

void fn(void) {
  for (int i = 0; i < 100000; i++)
    shared += 2;
  sema_up(&done);
}

int main(void) {
  sema_init(&done, 0);
  thread_new(&fn);
  sema_down(&done);
  printf("shared = %d\n", shared);
  return 0;
}
```

Listing 11: The program from Listing 7 synchronized with a semaphore.

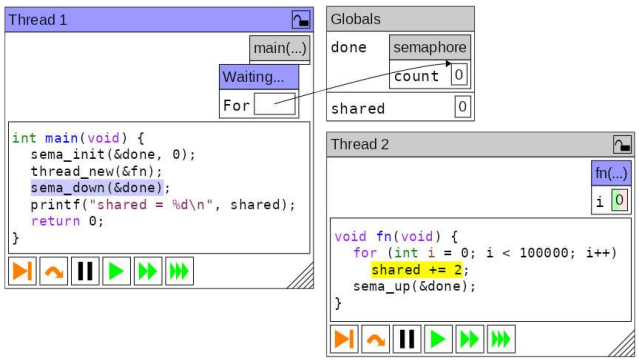


Figure 11: Progvis [82] used to visualize the code in Listing 11, which is a synchronized version of the code in Listing 7.

After introducing the semantics, the teacher asks students to use a semaphore to solve the issue in Listing 7 by modifying the code and testing it. This activity encourages students to make inductive and deductive inferences based on the formal semantics and their observations, and helps them to advance to stage U2 in the framework. Again, this doubles as stage A2 for the memory model. Students are likely to eventually arrive at a solution similar to the one in Listing 11, which works correctly.

Since concurrent programs are non-deterministic, testing programs for correctnes is difficult. Therefore, a visualization tool like Progvis is beneficial for this exercise as it allows students to single-step individual threads and observe the program state at each step. Figure 11 shows how Progvis illustrates a state where Thread 1 waits for the counter in the semaphore `done` to become positive. By observing that only Thread 2 is able to run and that it will eventually call *up*, it is possible to deduce that Thread 1 waits for Thread 2 to finish. Progvis [82] also detects and reports cases that are undefined according to the memory model, which allows students to validate and refine their understanding of the memory model. It is, however, worth noting that Progvis does not simulate *how* undefined operations might behave, which is why it has not been suggested until now. In Progvis it is also a good idea to reduce the number of iterations in the loops.

```c
int shared = 0;
struct semaphore done;

void fn(void) {
  for (int i = 0; i < 100000; i++)
    shared += 2;
  sema_up(&done);
}

int main(void) {
  sema_init(&done, 0);
  thread_new(&fn);
  for (int i = 0; i < 100000; i++)
    shared += 5;
  sema_down(&done);
  printf("shared = %d\n", shared);
  return 0;
}
```

Listing 12: The program from Listing 9 synchronized with a semaphore.

The previous example only illustrates waiting for another thread to complete its task. Listing 9 also illustrated cases where shared data needs to be protected. To illustrate the problem, the solution from Listing 11 can be applied to Listing 9, which results in Listing 12. In spite of the semaphore, the variable `shared` is modified concurrently and needs to be protected to ensure that the two memory accesses involved in each increment operation are executed as a unit.

How this can be achieved is likely not immediately clear to students, and thus encourages further exploration of stage U2 of the framework through deductive inferences based on their understanding of the semaphore. Hopefully, students will eventually realize that the goal can be re-stated as *waiting for other threads to not use the variable*. With this insight, it is possible to apply the previous solution here as well, which results in the solution shown in Listing 13, which correctly solves the problem. It is worth noting that the semaphore is used slightly different since the semaphore `mutex` is used to ensure *mutual exclusion*.

After students have found a solution, the teacher then helps them generalize the two solutions to help them move to stage U3 of the framework for *identifying* the abstraction of the semaphore. One way to achieve this is to imagine the counter in the semaphore as tracking the availability of some resource in the program. In this case, `done` tracks whether `fn` is finished or not, while `mutex` tracks how many threads may enter the *critical section* that modifies the variable `shared`. This insight introduces a higher-level concern that is useful as a strategy to guide students towards correct usage of semaphores. Applying it correctly does, however, require the ability to *move* between different levels of abstraction, and helps students move to stage U1 in the framework for that skill. In this

```c
int shared = 0;
struct semaphore done;
struct semaphore mutex;

void fn(void) {
  for (int i = 0; i < 100000; i++) {
    sema_down(&mutex);
    shared += 2;
    sema_up(&mutex);
  }
  sema_up(&done);
}

int main(void) {
  sema_init(&done, 0);
  sema_init(&mutex, 1);
  thread_new(&fn);
  for (int i = 0; i < 100000; i++) {
    sema_down(&mutex);
    shared += 5;
    sema_up(&mutex);
  }
  sema_down(&done);
  printf("shared = %d\n", shared);
  return 0;
}
```

Listing 13: The program from Listing 12 properly synchronized.

case between the level of the semantics of the semaphore and the higher level synchronization goals of the program.

```c
struct lock;
void lock_init(struct lock *l);
void lock_acquire(struct lock *l);
void lock_release(struct lock *l);
```

Listing 14: The public interface of a lock.

To help illustrate the difference between the different synchronization goals in the program, it is relevant to introduce the synchronization primitive *lock* (or *mutex*). A lock can be viewed as a special case of a semaphore that is designed for the explicit purpose of achieving mutual exclusion (like `mutex` in Listing 12). While this restricts its capabilities (e.g., it is not possible to replace `done` with a lock) it allows improved readability and error-checking. The interface provided by a lock is shown in Listing 14. The `lock_acquire` function corresponds to `sema_down`, and `lock_release` corresponds to `sema_up`. Discussing these differences further helps students' ability to *move* between different abstraction levels, helping them advance towards stage U2 in the framework.

Finally, to help students reach stage U3 of the framework, it is beneficial to have a discussion of code similar to the one in Listing 15. The code contains two functions, `a` and `b`, that both increment the global variable `shared` by 2 in three separate steps

```c
void a(void) {
  lock_acquire(&lock);
  int tmp = shared;
  lock_release(&lock);
  tmp += 2;
  lock_acquire(&lock);
  shared = tmp;
  lock_release(&lock);
}

void b(void) {
  lock_acquire(&lock);
  int tmp = shared;
  tmp += 2;
  shared = tmp;
  lock_release(&lock);
}
```

Listing 15: Illustration of the importance of critical sections.

(corresponding roughly to the low-level implementation of +=). The two functions have both been synchronized using a lock: `a` only holds the lock when `shared` is accessed, while `b` holds it throughout the entire increment operation. The teacher asks students to find problems if two (or more) threads execute `a` and/or `b` concurrently. The goal is for students to realize the importance of higher-level synchronization goals. In this case, both versions are well-defined according to the memory model (i.e., no data races). However, if `a` is used, an issue similar to what happened in Listing 13 might occur. As such, this further helps students develop their ability to *move* between different abstraction levels. In this case, students must move between the program level (considering goals) and the level of individual operations that achieve these goals to determine which operations must execute as a unit, helping them to arrive at stage U3 in the framework for this skill.

5.3.5 Course Content: Creating Abstractions. The final concept is *creating abstractions*, where students apply their knowledge of the *memory model* and *synchronization* together with their knowledge of abstractions in sequential programs. This allows them to generalize their ability to create abstractions to include the additional concerns that are relevant in concurrent programming. This part of the course is therefore exclusively related to the *creating* skill of the proposed framework.

These aspects can be illustrated by starting from an implementation of a simple stack, such as the one in Listing 16. Initially, the stack is not synchronized, since it does not protect its shared data using synchronization primitives.

To prompt students to explore these new properties, thereby working with stage U1 in the framework, the teacher shows the stack implementation from Listing 16 alongside a main program (e.g., the one in Listing 17), and asks students to synchronize it. Assuming that students have advanced far enough in the previous topics, they should be able to deduce that the variable `count` is shared and needs to be protected (e.g., using locks).

There are two main approaches to achieve this. Either placing locks in the *implementation* of the stack, or by placing locks in the

```c
struct stack {
  const char *elements[MAX];
  int count;
};

bool stack_push(struct stack *s, const char *e) {
  if (s->count < MAX) {
    s->elements[s->count++] = e;
    return true;
  } else {
    return false; // Full.
  }
}

const char *stack_pop(struct stack *s) {
  if (s->count > 0) {
    return s->elements[s->count--];
  } else {
    return NULL; // Empty, signal error.
  }
}
```

Listing 16: Implementation of a simple stack in C.

```c
struct stack s;

void other(void) {
  stack_push(&s, "a");
}

int main(void) {
  stack_init(&s);
  thread_new(&other);
  printf("Popped: %s\n", stack_pop(&s));
  return 0;
}
```

Listing 17: Example of a main program that illustrates problems with the stack implementation in concurrent programs.

code that *uses* the stack. Ideally, students use both approaches when solving the problem in the class so that the teacher can highlight the difference between their solutions. In particular, the former approach makes the stack *thread-safe* since it is safe to use from multiple threads concurrently. This helps students move towards stage U2 in the framework.

Even though adding locks to the stack avoids race conditions, the program in Listing 17 does not behave as intended. Since the thread that pops an element does not wait for other threads to push an element, the pop might fail and return `NULL`. A convenient solution to this problem is to use a semaphore to make `stack_pop` *wait* until there is an element to pop. The key insight here, which further helps development of stage U2, is that abstractions in concurrent systems also have the option to *wait* for something, just like the synchronization primitives. In particular, `stack_pop` could either return `NULL` or *wait* until an element is available. In this case, the latter makes the example program behave as intended.

Finally, to help students to reach stage U3 of the framework, they need to consider (potentially with the help of the teacher) where synchronization primitives should be declared. In this case, there are essentially two options: either inside the data structure or globally. For the locks, either option works, but the latter is suboptimal as it needlessly disallows different stacks from being used concurrently. It is not possible to declare the semaphore globally, as its counter represents the number of elements in the stack. If multiple stacks are used simultaneously, they would therefore not behave as expected. These problems are, however, not shown by the simple example program and thereby benefits from discussions with a teacher.

5.3.6 Course Content: Summary. One observation from above is that there is a common theme between the different stages. In stage U1, we first try to illustrate the problem or the concepts that should be learned through an example that the students can use to experiment later on. In stage U2, students are then asked to use the example to experiment and observe its behavior in order to be able to explore the concepts. However, due to the non-deterministic nature of concurrency, it is essential to reason about the behavior of the system in relation to the observations, and in particular to determine which behaviors are undefined and which are defined. Finally, in stage U3, we see exercises where students need to use what they have learned so far to reason about more general cases, or cases that can not be easily or reliably observed in practice.

Also, this illustrates the interleaving of understanding a new concept with applying previous steps, so that learning the new concepts acts as a process of learning to apply the previous concept. This is, of course, not the case for the last concept: creating abstractions. This part thus has to be further practiced through additional exercises.

It is also worth noting that the examples above illustrate the overall progression in the course with relevant examples. There are, of course, more examples at each level that students can use to practice. In particular, the course contains a series of lab assignments where students apply these skills in a larger codebase.

5.3.7 Practical Considerations. The outline for teaching concurrency presented in this context is used to teach concurrency in the course outlined in Section 5.3.1. However, the exact code examples are more specific to operating systems, and the *ordering* of the stages differ. The stages are reordered to ensure that students have enough background to start working with the synchronization assignments in Week 3. This means that, rather than introducing the three main topics one by one in their entirety, they are instead introduced iteratively, in small pieces. That is, the three concepts above are covered multiple times at different depth. The drawback of this approach is that it is difficult to have a deep discussion of the more advanced concepts initially. For this reason the first iteration only focuses on the *memory model* and *synchronization*, leaving *creating abstractions* for later iterations. There are, however, advantages to this approach. First and foremost, it lets students advance to a stage where they are able to use synchronization primitives to experience earlier. Secondly, it naturally leads to repetition of all concepts in the course, which Armoni [6] noted is important.

In spite of the drawback mentioned above, the iterative approach works well in practice. As such, this approach is an option in courses that involve a number of dependent concepts, and where all concepts are necessary for students to be able to start solving interesting assignments. Another option would be to re-design the assignments to be better aligned with the *idealized* ordering above. This has been done to some extent by adding introductory assignments in Progvis, but it is not practical to implement completely.

5.4 Context 4 - Recursion in Secondary Education

5.4.1 Context of the Field. Recursion is a fundamental technique in computer science, but is considered difficult for novices to understand [4, 50, 92]. The source of this difficulty has been attributed to a wide number of specific misconceptions that have been identified and modelled by various authors [36]. According to some studies, students tend to develop highly divergent and idiosyncratic models of recursion and "speak a very different language from that of the experts" [51, pp. 306] when it comes to the topic, indicating that novel pedagogical approaches for recursion education can potentially be impactful.

5.4.2 Context of the Module. This context outlines a module for secondary school students that could be introduced as a standalone workshop or integrated into a course, for instance one that prepares students for the Advanced Placement (AP) Computer Science A examination in United States, in which recursion understanding is tested [16]. Following the same design as a recursion workshop described in a 2024 paper [86], participants will be instructed in recursion using the visual programming language Algot [87, 93], a recursion-based programming language which has been tested in controlled studies in secondary and tertiary education settings [35, 86, 88, 89]. In Algot, the program state is always visible, which may help students understand the transition from concrete examples to abstract concepts by visually tracing how specific values change as an operation is executed. By observing how the the operation behavior changes under different inputs and input categories, students are expected to have a better foundation for generalization.

5.4.3 Application of Abstraction Skills within this Context. Decades of research has studied the specific problems that students experience with conceptualizing and applying recursion. Hamouda et al. [36] composed a list of problematic aspects of recursion that have been described frequently in the literature, for example the *passive* and *active control flow* of a recursive function after reaching the base case [32], the *limiting case* [69] (meaning how to formulate and trigger a stopping condition), and comparisons to loops [7]. Generally, in order to implement and understand recursive problems, students must manage the relationship between the whole problem and its parts, which means identifying the underlying structure, isolating the repetitive elements, and defining a stopping condition. To do so, students must apply abstraction skills, meaning that they must understand which aspects of a problem can be separated into self-similar subproblems and to discard the elements of the problem that are not relevant, for example how the recursive algorithm handles specific input values.

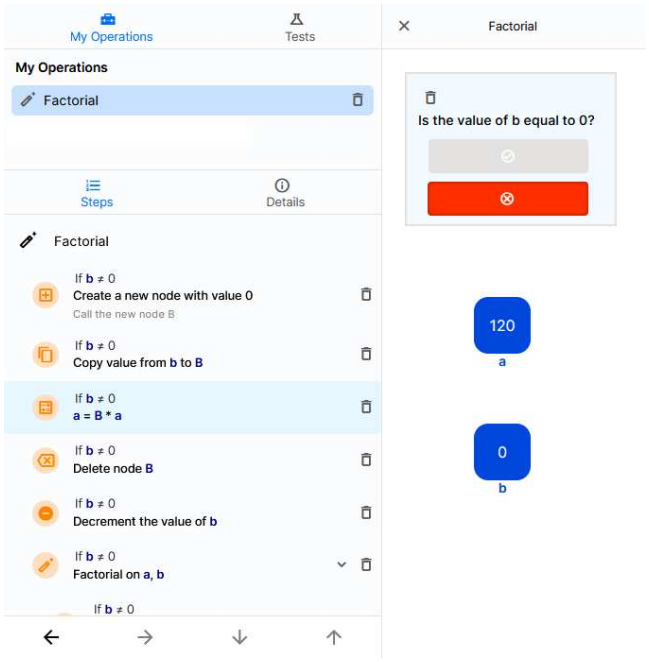


Figure 12: Computing a factorial in the visual programming language Algot using concrete example values.

5.4.4 Applying our Framework. Using our model of concept formation, we propose the following application of our framework for teaching recursion at the secondary school stage:

- **Understanding**

U1: Observation: We propose instructor-led, live coding activities, which are considered one of the best practices for teaching programming [73]. Students will be shown existing programs in Algot for demonstrating recursion, such as a factorial (see Figure 12) or tree-based algorithms. In this stage, Algot is used mostly as an algorithm visualization tool that shows how specific values change throughout the execution of the program. The instructor can use Algot to address common misconceptions about recursion, for example by using (i) the step-through functionality to explore demonstrating backward and forward control flow within recursive calls and (ii) the query panels for demonstrating the control flow of the program, showing how a stopping condition can be defined, for example.

U2: Inductive and deductive inferences: Using Algot as a level four algorithm visualization tool according to the taxonomy by Naps et al. [60], the teacher will ask students to explore alternative input values for the recursive programs that are under consideration. The teacher will prompt students to consider which parts of the program stay the same and which change as the input values differ. This way, the teacher can help students in making inferences about the mechanics of the algorithm that is being considered; for instance, for the factorial function, the students may observe that the algorithm will always reach the stopping

condition no matter how large the input value is, but that the output grows rapidly.

U3: Hypothetical thinking: In a group-based problem solving sessions, students are asked to consider how the problems they were shown generalize onto other inputs, for example, what are reasonable preconditions, what branches are explored for certain classes of inputs, and how the structure of recursive code differs from the structure of iterative code.

At this stage, students formulate hypotheses about the behavior of recursive algorithms beyond the examples provided. They participate in group-based problem-solving sessions where they discuss the outcomes of edge cases (such as when a recursive function receives negative, zero, or extremely large input values), first by predicting the function's behavior and then by testing their hypotheses using Algot, possibly observing outcomes like infinite recursion or other errors. Using the live functionality of Algot, students will be able to trace their programs using specific input, thereby understanding not only what a function returns, but why.

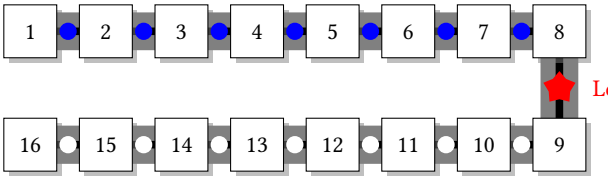
Students will also identify and discuss the structures in recursive solutions, such as divide-and-conquer strategies, and consider how these patterns apply to different types of problems. Ultimately, students are expected to be able to write their own recursive algorithms when given a problem that they have not seen before. This progression indicates that the students have moved beyond specific examples to a generalized understanding of recursion, and in this way, have met the goals of the framework by being able to apply abstract concepts to unfamiliar situations.

- **Application**

A1: Observation: In this direct instruction module, we will use Algot as a programming language as opposed to a static algorithm visualization tools. The instructor will describe specific, broad problem domains and explain how they can be solved in recursion, followed by implementing these solutions in Algot.

A2: Inductive and deductive inferences: Algot will be used as a programming language for hands-on, student-led activities, implementing recursive solutions "by recipe"; students are given problems where specific recursive algorithms are first explained in plain language using real-life examples, after which students are then asked to implement them in a skeleton environment in which appropriate example input values have already been given. We propose (i) unplugged activities that have been introduced at the lower secondary [53] such as calculating integer powers of two with recursion, using group communication and Lego blocks, (ii) secondary school programming competition problems like Bebras Challenge problems, such as the binary search problem shown in Figure 13[12], and (iii) tree-based recursion problems that have been tested in Algot at the secondary level such as assigning every node in a binary tree a value that is proportional to its depth [86].

A3: Hypothetical thinking: Using Algot's extension for testing [31], we propose a test-driven approach for generating hypotheses about the programs, verifying whether the student-implemented algorithms are correct, inviting students to invent edge cases and to test their solutions accordingly. Test-driven approaches support hypothetical thinking by allowing students to measure their beliefs or preconceptions about their code against reality.



There is one (and only one!) leak in the neighborhood terminal pipes. The plumber has asked all the homeowners to turn off the water in their homes. She then closes the valve between two houses and checks the water meter to see if the leak is between those two houses. For example, if by closing the valve between house 8 and 9 the meter indicates water flow (as shown in the picture), the plumber knows that the leak is certainly between house 1 and 9 (and there are no leaks between 9 and 16). How can the plumber find the leak by checking as few pipes as possible?

Figure 13: A modified version of a recursion problem from the The Bebras International Challenge on Informatics and Computational Thinking.

5.4.5 Summary and Comparison. In this context, we show how our proposed context fits to a course that guides students through a progressive learning path that begins with concrete observation and moves towards abstract understanding. Initially, students trace recursive algorithms in Algot, working with specific examples on trees and linked lists, followed by engaging in inductive and deductive reasoning by experimenting with different input for recursive functions. Lastly, students employ hypothetical thinking to generalize the patterns they have learned onto new and unfamiliar problems.

We also note that this approach is well-aligned with the principles of Concreteness Fading [28], where learning also transitions from concrete representations to abstract concepts. This helps to illustrate the connection between our proposed framework and existing frameworks. Algot's visual programming environment supports this transition by providing concrete, live examples that can be discarded as students progress toward more abstract representation.

5.5 Context 5 - Teaching Linear Data Structures in CS2/CS3

5.5.1 Context. The context discussed in this section introduces linear data structures in a CS2 course at a US university. The course

[12]See https://bebras.it/platform/html/player_teacher.html?class=screenshot&code=2018_Q27

involves 3 hours of live programming lectures over 3 days in a week across 15 weeks of semester. There is a 1 hour lab session each week to further enhance the understanding of concepts explored during that week. The topics start with a general review of memory management in C programming language and quickly escalate to linear data structures, sorting algorithms, complexities to recursion, non-linear data structures, searching algorithms etc.

5.5.2 Prerequisite Expectations. This course follows from a procedural programming course (CS1) in C programming language. Singly Linked List (SLL) is the second data structure introduced where data is stored in an ordered way. The first linear data structure introduced is a simple array/list. Leading up to the discussion on singly linked lists, the students work on smaller exercises with pointers and reference manipulation exercises to prepare.

5.5.3 Activity: Motivation. The students often look for motivation when a new concept to solve a problem is introduced and a solution to the problem already exists. A good motivation exercise keeps the students invested in the learning process from understanding the problem to applying it. The question that comes up every time this topic is introduced is - "Since there is an array to store data in an ordered manner, why do we need a linked list at all?" To answer this question, we do a simple exercise in the classroom. We ask the students to assume that the classroom with students is a computer's memory and the seats occupied by students are unused/available memory. Now the teacher ask the students to allocate an array of size 5; i.e. we need to find 5 students sitting next to each other since arrays need consecutive free space to be allocated. We next increase the array size to be allocated to 10. The number of available options goes down because it is now more difficult to find 10 students sitting next to one another. When the array size goes up to 20, we mostly have one set of 20 students sitting together. So, we establish that as the size of the array goes up, it is increasingly difficult to find consecutive free memory space to allocate to the array. This leads us to establish the need for a data structure that can hold large amounts of data efficiently and hence the motivation for singly linked lists. This example therefore uses the *move* abstraction skill at levels U1 and U2 in the framework.

5.5.4 Activity: Building Blocks. In order to understand a singly linked list, the students need to have a good understanding of the building blocks that make the list (and thereby developing the *identify* abstraction skill). So, the structure of a single node of the singly linked list is first introduced to the class. The students at this point are simply in stage one of cognition of the proposed framework where they make an observation of the code presented to them. The code in Listing 18 is presented to the class and is self referential. The students are encouraged to discuss and comment on whether the code would compile. The students try to induce and deduce an answer from their previous knowledge of code syntax moving them to stage U2 of the framework. There is usually a small percentage of the class that says the code presented in Listing 18 would not compile. Their argument states: "How can you define a `next_node` of the type `sll_node` within the definition of `sll_node`?" At this time, we move to stage U3 of the framework by writing the code to construct the node and execute it to prove that the presented

```
// Step 1 - Create the struct for sll_node
typedef struct sll_node{

  // Step 2-1 - Create a int to hold data
  int data;

  // Step 2-2 - Create a struct to hold pointer
    to next node
  struct sll_node* next_node;

}sll_node;
```

Listing 18: Example of a code defining the structure of a single node.

```
sll_node* insert_node = (sll_node*)malloc(
    sizeof(sll_node) );

insert_node->data = 100;
insert_node->next_node = NULL;
```

Listing 19: Example of constructing a single node, allocating memory and updating data/reference variables.

code in Listing 18 is in fact correct. We create a reference to a single node and allocate memory space to it in the main method. We further assign values to the data and the `next_node` reference as shown in Listing 19. The code executes resolving any reservations about it that students had. The students confirm their understanding through application and we can move to the next concept. By passing through the three stages of cognition, the students develop the proposed framework's outcome skill of "Understanding".

Since the students are now in acceptance of the self referential code for a single node, we move towards a diagrammatic representation of the single node in memory. We call these *memory diagrams* and they help create a visualization of the code. The memory diagram for code in Listing 18 and 19 is given in Figure 14. The students once again start with the framework's stage U1 of observation but this time to develop an understanding of a different abstraction (memory diagrams). The students make an observation of the nuances of representing a memory space, reference pointer and labeling variable names and such through this exercise. The students ask questions to learn about how the primitive data is stored versus a variable that holds a reference.

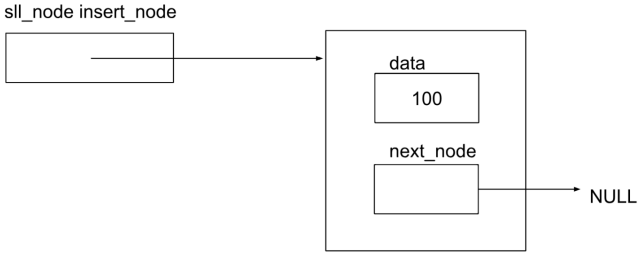


Figure 14: Memory diagram of a SLL node

The students move to the next stage of cognition, U2, where they induce from their observation of a single node and deduce how that can be extended to the idea of a singly linked list that is a list consisting of multiple list nodes using the memory diagram approach as shown in Figure 15. The students deduce how to make reference manipulations through a diagram and gain an understanding of the second abstraction presented to them. The memory diagram helps establish the reference connections that need to be constructed for successfully building a linked list. As they move to the final stage of the framework, U3, the following statements are made by the instructor to help the students observe the properties of a SLL

- A reference is needed to hold on to only the first node in the list,
- Each internal node's `next_node` reference must point to the next node in the list,
- The `next_node` reference of the very last node in the list must point to `NULL`.

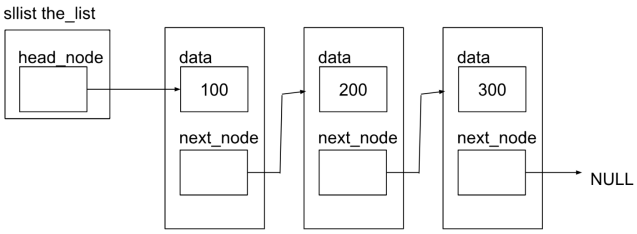


Figure 15: Memory diagram of a SLL with multiple nodes

At this point in time, we can assume that the students are acquainted with the two abstractions presented to them: the code and the diagram for the idea of a singly linked list. We write the code for a singly linked list that consists of the handle/reference to the first node or the head node in the list of nodes as given in Listing 20. We continue to write the code to allocate space for the

```
// Step 3 - Create the struct for sllist
typedef struct sllist{

  // Make the sll_node member called head_node
  struct sll_node* head_node;

}sllist;
```

Listing 20: Example of code defining the structure of a SLL.

singly linked list as shown in Listing 21 within the main method. The students' attention is drawn to observe that the (empty) list initially has its `head_node` reference point to `NULL`.

Once the students have been introduced to the idea of a single list node and singly linked list both visually using the memory diagrams and code, they need to utilize, and thus develop, the abstraction skill of moving between different abstraction levels.

We move on to the next set of topics that deal with how to manipulate (insert, access, update or remove) data within the list. In teaching those topics, which will also help in assessing whether

```
// Step 4 - Dynamically allocate a sllist
sllist* the_list = (sllist *)malloc( sizeof(
    sllist ) );

// Step 4-1 - Set the head_node to NULL
the_list->head_node = NULL;
```

Listing 21: Example of code constructing a SLL, allocating memory and updating data/reference variables.

students have reached the stage 2 subgoal of understanding how to move between abstraction levels as it relates to the concepts of a single list node and singly linked list.

5.5.5 Activity: Inserting into SLL. In this section, the students are in stage one of the framework where they observe the entire process of writing the code to insert into an empty list and to the end of the list. The students are expected to observe and gain understanding of reference manipulations done in code and via memory diagrams for the process of adding a node to the list. They will be later be expected to draw from their understanding to apply to a different kind of list manipulation. The next exercise introduces the reference updates that are needed to construct and insert a new node into a SLL. A new function is added to insert a node into the list. The function parameters include the reference to the SLL `the_list` that was constructed in Listing 21 and an integer value that needs to be stored in the new node. The `insert` function definition is given in Listing 22.

There are some special cases that need to be considered when adding a new node to an SLL that are discussed in the following sections.

```
void insert( sllist* the_list, int the_value ){
  //Code to insert a list node goes here!!
}
```

Listing 22: Function definition of insert function

A. Adding a new node to an empty list: This is the simplest case to add a new node to the SLL. First, a new node (`insert_node`) must be constructed with memory allocated and data and `next_node` references updated to the integer value (passed in as a parameter) and `NULL` respectively. Next, there needs to be a check in place to infer that the SLL is in fact empty. If the list is empty, the head node reference points to `NULL` (the same as when `the_list` reference was created above in Listing 21). If the list is empty, `the_list` reference's `head_node` reference must be updated to point to the new node. The example code to insert into an empty list is given in Listing 23. We write the code alongside the students in a live programming exercise. The students are creating code and observing the code writing process while doing so. The students are asked to draw the memory diagram for an SLL with a single node. The three statements (listed above) defining the properties of an SLL are revisited by the instructor. The students then are asked to deduce

if the properties of an SLL are satisfied by their memory diagram.

```
1 // Create a sll_node pointer called insert_node
2 sll_node* insert_node = (sll_node*)malloc(
      sizeof(sll_node) );
3
4 // Set the insert_node's data to the_value and
      next_node to NULL
5 insert_node->data = the_value;
6 insert_node->next_node = NULL;
7
8 // Check if the list is empty
9 if( the_list->head_node == NULL ){
10
11     // Set the head node to point to insert_node
12     the_list->head_node = insert_node;
13
14     // And return
15     return;
16 }
```

Listing 23: Example code to insert into an empty list

B. Adding a new node to the end of the list: In a singly linked list, the common place to add a new node by default is at the end of the list. The code for the previous case (add to empty list) can be extended to add to the end of the list. We can continue after line 16 in Listing 23 by adding an else block that will contain the code for adding to a non-empty SLL. If the list is not empty, we need to traverse to the end of the list and update the `next_node` reference for the last node in the list to point to the new node. The code for adding a new node to the end of a non-empty SLL extends from the code above as given in Listing 24. We continue the process of live coding where the students create code in parallel and the observation of the code continues and possibly moving into the stage two of deducing from the code they have been typing in so far.

```
1 // Otherwise, create a curr_ptr reference to the
      head_node
2 sll_node* curr_ptr = the_list->head_node;
3
4 // Iterate until the next_node is NULL
5 while( curr_ptr->next_node != NULL ){
6
7     curr_ptr = curr_ptr->next_node;
8
9 }
10
11 // Set curr_ptr's next_node equal to insert_node
12 curr_ptr->next_node = insert_node;
```

Listing 24: Example code to insert to the end of list

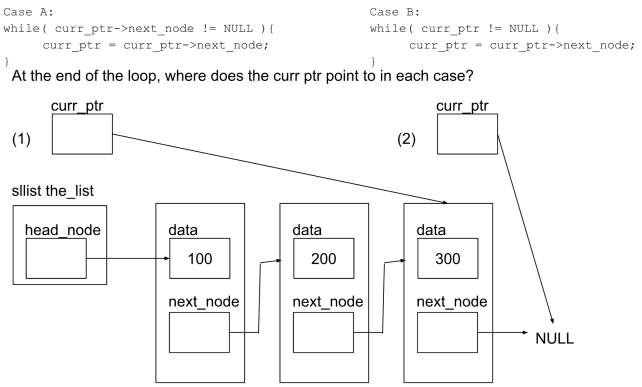


Figure 16: Exercise to understand the stopping point for traversal

One aspect of the process of inserting to the end of an SLL is traversing the list. It is very important for the students to capture the essence of traversing the list in entirety because that concept is reused multiple times in list manipulation (update and remove functions). It is also a concept that a number of students have trouble getting right. We dive deeper into developing the understanding skill of the traversal process with the help of an exercise. The students are expected to read the code in Listing 24, trace through the code if the condition in the while loop is changed to:

```
while( curr_ptr != NULL )
```

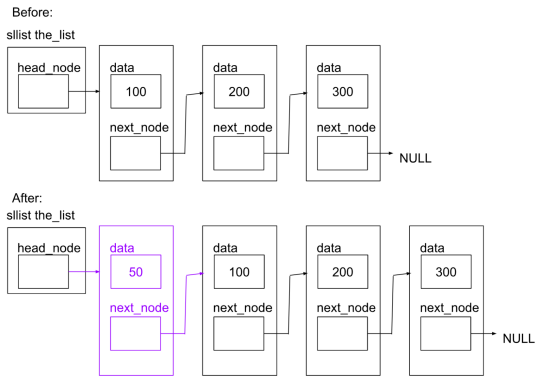
instead of

```
while( curr_ptr->next_node != NULL )
```

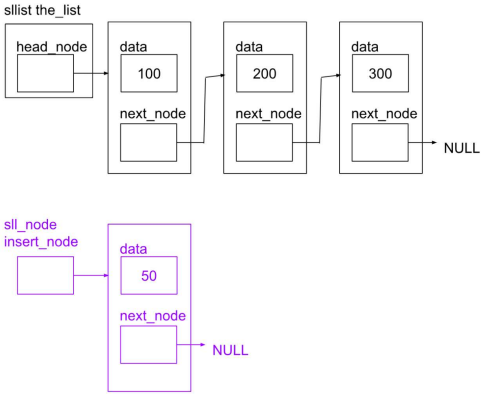
The students are then encouraged to move to stage U2 in the framework and deduce through code tracing and map the code to the correct memory diagram without executing the code. The students are required to map the `curr_ptr` reference in Case A and B to (1) or (2) in the Figure 16. Once they have selected an answer, they are asked to execute both sets of code and record the result. When executed, Case B results in a segmentation fault and shows the students what the stopping condition needs to be so that the `curr_ptr` reference does not fall off the end of the list. We re-emphasize the importance of understanding how far to traverse in the list as the list traversal is a technique that is a directly transferable concept when removing or updating in a SLL. Finally, to help students reach stage U3 of the framework, they are asked to trace the code and develop an understanding of the traversal process.

The students are encouraged to draw the memory diagram of the SLL after a node has been added to the end of the list. The students use their understanding of empty list and SLL memory diagrams and take their skills to move between abstraction levels from the previous exercises to create a memory diagram for adding a new node to the end of the list.

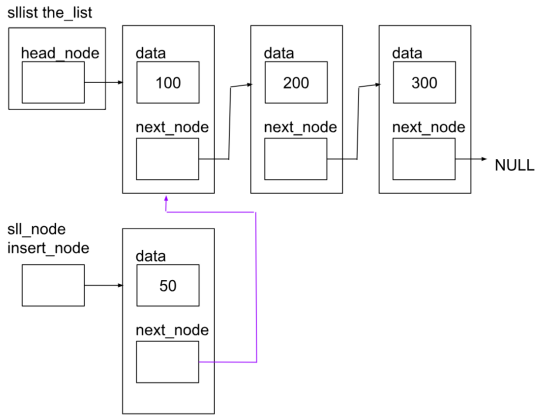
In a singly linked list, adding to a list by default translates to adding to the end of the list. However, in order to solidify



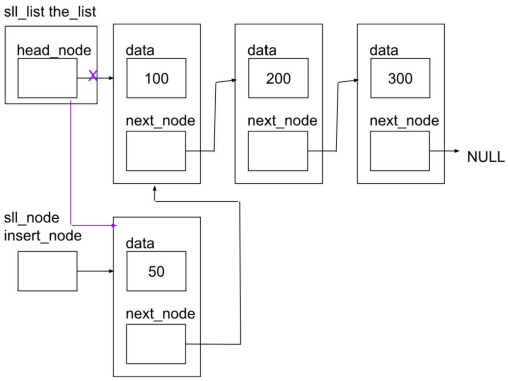
(a) Before and after memory diagrams to add a new node to the front of the list



(b) Step 1: Construct a node, allocate memory, update data and next node reference



(c) Step 2: Update `next_node` reference of new node to point to first node in `the_list`



(d) Step 3: Update `head_node` of `the_list` reference to point to the new node

Figure 17: The steps involved in adding a node to the front of a linked list.

students' understanding of the reference updates further, exercises to add to the beginning and middle of the list are also explored as explained in the following sections.

C. Adding a new node to the beginning of the list To insert a node to the beginning of a list, a new function is required as adding to the front of a SLL is a specialized function and ordinarily not a default function of a SLL. The function definition for inserting at the beginning of the list is given to the students as shown in Listing 25. The students are given

```
1 void insert_front( sll_list* the_list, int
      the_value){
2   //Code to insert at the front of the list goes
      here!!
3 }
```

Listing 25: Example code to insert to the front of list

before and after memory diagrams and asked to write the code for the reference updates needed to achieve the changes as shown in Figure 17a. The students are expected to make their observations and deducing from what they have seen so far.

Following the before and after memory diagrams, students are given the intermediate memory diagrams of each step required to get to the final diagram. This scaffolding helps students narrow down their observations to move between abstraction levels of memory diagram to one line of code. The changes for Step 1 are shown in Figure 17b. The students are expected to observe the memory diagram and hypothesize that they need to construct a new node to create the line of code for this step.

The changes for Step 2 are shown in Figure 17c. The students are expected observe that `next_node` reference of the new node needs to be updated to point to the reference that is pointed to by the `head_node` of `the_list`. This aims to develop their abstraction skill (create) to write appropriate line of code to complete this step, in particular by developing

the outcome skill of application (A1 and A2) in this and the following steps.

The changes for Step 3 are shown in Figure 17d. The students should observe that the head_node reference for the the_list must be updated to point to the new node and write the line of code for this step.

To finish off the exercise, the students are required to discuss and answer if the insert would work as intended if the steps 2 and 3 are swapped (i.e., Step 3 is executed before Step 2.) This exercise helps the students to solidify their outcome skill of understanding. They will have to use the *identify* abstraction skill to use, make observation of the before and after diagram, deduce that swapping the lines in the code would not work. They should be able to redraw the memory diagrams and explain that if Step 3 is performed first, the handle to the list (with nodes 100, 200, 300) is lost.

D. Adding a new node to the middle of the list: A new function is required to add to the middle of the SLL as the location of insertion needs to be passed in as a parameter to the function. The function definition for inserting in the middle of the list is given to the students as shown in Listing 26. The students are given a before and after diagram for inserting into the middle of the list as shown in Figure 18a.

```
1 void insert_middle( sll_list* the_list, int
      the_value, int index ){
2   //Code to insert at index in the list goes
      here!!
3 }
```

Listing 26: Example code to insert in the middle of list

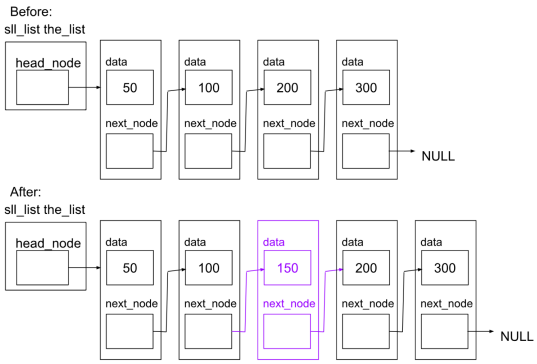
The key steps in achieving the final list would need the steps in Figure 18. The changes for step 1 to 4 are shown in Figures 18b to 18e.

The exercise to add to the middle of the list is very similar to the previous exercise to add to the front of the list. This helps students to hone their outcome skill of understanding while developing their abstraction skills to identify similarities between the previous exercise, working on their moving skills between abstraction levels of memory diagram and code and practicing their skills to create small snippets of code.

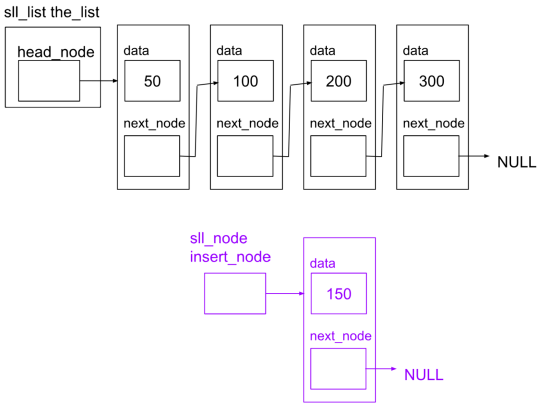
As a final exercise, the students are asked to answer the following two questions:

- "In Step 2, why do we stop traversing at the node before insertion index?"
  The discussion for this question leads us to the motivation for the next topic to come in instruction - a doubly linked list. In a singly linked list, one can only traverse in the forward direction. We stop traversing at `index - 1`, so that we can connect `next_node` reference of the new node to the node at `index` and the `next_node` reference of the node at `index - 1` to the new node (in that order).
- Would switching steps 3 and 4 produce similar result? This question start the students thinking on how to achieve a similar result with a different approach. In trying to manipulate references in a different order, their understanding and confidence in applying their previous learning gets them more prepared for solving the next exercise with minimal support.

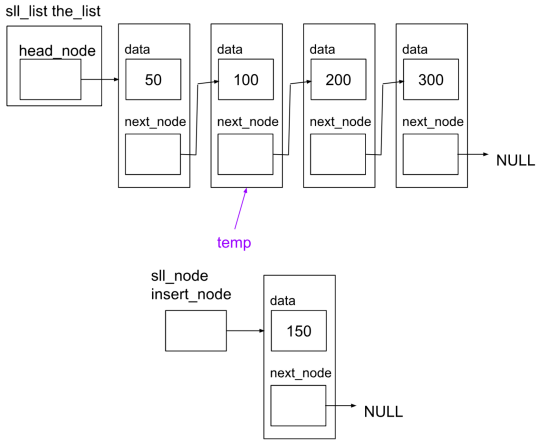
5.5.6 Activity: Removing from SLL. Finally, the students are given an exercise to remove a node from the list. At this point, the students are given a prompt in the natural language with no intermediate support with memory diagrams to get to the final code. The students are expected to apply their understanding of reference manipulations from the previous sections and generate the before and after memory diagrams for the remove action. They need to produce the intermediate memory diagrams and the final remove functions to remove from the end of the list, beginning of the list and middle of the list. This exercise aims to help students to develop their application outcome skill and go through the stages of observing, deducing and hypothesizing and developing their abstraction skills to identify, move between abstraction levels and create code from all the knowledge they have gained for supported practice exercises so far.



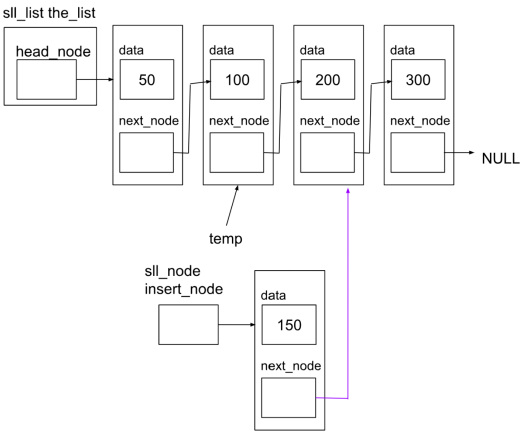
(a) Before and after memory diagrams to add a new node to the middle of the list



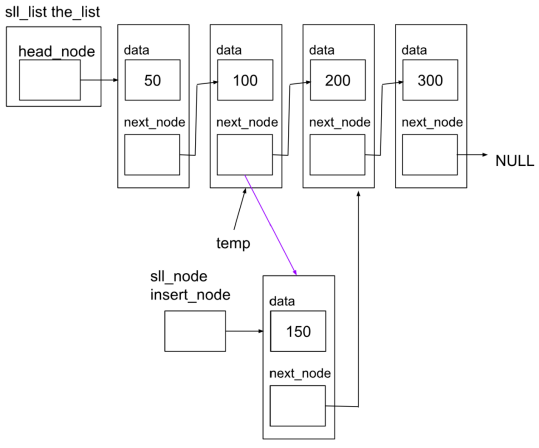
(b) Step 1: Construct a new node with the data parameter and `next_node` set to `NULL`



(c) Step 2: Define a counter and iterate until the counter reaches one less than the required index. Store the reference to the last node visited in a temporary variable



(d) Step 3: Update the `next_node` reference of the new node to the next node of the temporary variable from previous step



(e) Step 4: Update the `next_node` reference of the temporary reference to the new node

Figure 18: The steps involved in adding a node to the middle of a linked list.

6 Discussion

This section summarizes insights from applying the framework to the contexts (Section 5). These insights were then used to complement the final framework with insights of how the framework may be applied. This final version of the framework is presented in the next section (Section 7). We also discuss how our framework differs from other systematizations, present the limitations of our approach, and propose avenues for future work.

6.1 Fitting the Framework to Contexts

Applying the framework to the contexts initially resulted in much discussion within the group, mostly regarding which abstraction skills were actually used at different stages of concept formation, and whether the distinction between inference types and core abstraction skills were useful. These discussions led to multiple revisions of the framework. The final version presented in this paper is thus the result of an iterative design process and we are confident that the framework is able to adequately describe the activities within all of the contexts. The wide range of topics covered in the contexts in Section 5 also helps verify that the framework is general enough to suit a wide range of abstraction-related topics.

One thing that is noteworthy, however, is that the contexts take different approaches to use of the framework, ranging from analysis of existing teaching to proposals of new teaching and learning activities.

In Context 1, the framework is used to evaluate existing teaching of a specific low-level concept and identify opportunities for improvement, demonstrating the framework's usefulness as a practical tool for evaluating and planning teaching activities. The context relies on the well-documented pedagogical device of a notional machine. As such, it illustrates that it is possible to use such pedagogical devices within the framework. It is also worth noting that this context purposefully focuses on the first and second stages of concept formation, leaving the third and most advanced stage of concept formation to later activities and courses.

In Contexts 2 and 3, usage of the framework was structured around the activities in the course—for each high-level activity in the course, the instructor described which abstraction skills were involved at each stage, and which sub-goals of the activity were used to help the student develop certain skills. These two contexts also demonstrate the versatility and effectiveness of the framework in specialized, advanced areas of computer science in addition to the more foundational topics considered in the other contexts. In Context 2 in particular, the framework aided the instructor in structuring a large and complex topic into a suitable progression for students. In this case, the framework highlighted the reasoning students needed to make at each stage of concept formation, and what kind of questions an educator can use to achieve them. Similarly, Context 3 illustrated how the *understand* and *apply* outcome skills are related, especially in a larger module.

Context 4 is a proposal for a new unit of teaching, designed according to the framework. In this context, the framework is used to ensure that the proposed activities are appropriately staged and that both outcome skills are targeted. This context also illustrates the leap in inferential deductions between stage 2 and 3 of the framework (i.e., the difference between condensation and reification in the Sfard model [75]) for recursion. In particular, it shows that in stage 3 students understand the purpose, mechanism, and implementation of a concept, and are able to relate it to other concepts.

Context 5 is perhaps the most formal and bottom-up application of the framework. Each low-level learning objective is considered separately and the abstraction skills involved are identified. This approach leads to the development of specific activities (discussion prompts, coding demonstrations) that are informed by the students' expected stage of concept formation and the abstraction skills involved.

The range of different approaches to the use of our framework highlight its flexibility and adaptability to different contexts and purposes.

6.2 Insights from Applying the Framework

Although the framework is flexible, applying the framework to different contexts produced insights about teaching activities that are suitable at different stages, as well as the ordering of different but intertwined concepts.

6.2.1 Suitable Learning Activities. In the contexts, we have seen a large number of activities being used to help students advance through the stages of the framework. A common theme is that students need to be active at least in the learning activities that aims to develop stages 2 and 3 of the framework. How students are active, however, differs between contexts. In some contexts, students are activated through questions from the teacher during lectures, and at other times through assignments or parts of assignments that students solve on their own, or with the help of a TA.

As mentioned, for stage 1, the situation is a bit different. It is typically associated with learning activities where students do not need to be as active, because the main focus in stage 1 is to observe and make inferences on what is directly observable. As such, there are many situations where stage 1 is covered during a lecture, through introductions or questions from a teacher.

Since the learning activities that are suitable will necessarily vary depending on the concept being learned, the framework only offers guidance by framing the content in the stages and core skills presented in the framework. Due to the desire for the framework to be general enough to encompass multiple concepts, it is not able to prescribe particular activities apart from this framing.

6.2.2 Ordering Concepts, Stages of Concept Formation, and Abstraction Skills. In spite of the linear nature of some of the systematizations in Sections 2 and 3, our framework does not impose a strict linear progression in general. The only constraint is that for a particular concept and a particular abstraction skill, the student needs to progress through the three stages of concept formation in sequence. Apart from that, it is left up to the educator (or the student) to move across the framework as they see fit.

Contrasting examples of different approaches to ordering concepts and skills can be found in Context 3. This context initially proposes an idealized linear progression, allowing students to progress through all three stages of concept formation with respect to one concept before moving on to the next. This is beneficial since it encourages students to have a complete understanding of the prerequisites before building further skills (in this case, learning the

memory model before starting on synchronization). While this avoids situations in which students make incorrect inferences on higher abstraction levels, thus improving self-efficacy, it has the downside that it does not re-visit previous stages to reinforce learning, which Armoni [6] noted is important.

Towards the end of Context 3, an alternative structure is proposed, driven by the need to prepare students for the start of the computer lab assignments. As such, this structure visits the first two stages for one concept (in this case, the memory model), followed by the first stage for the next concept (in this case, synchronization). This means that students arrive at a point where they have a workable understanding of synchronization, which allows them to start working independently on lab assignments and advancing their understanding through independent practice. Future lectures are then used to re-visit these concepts to further develop them so that students can eventually reach the final stages for each of them. This gives students time to make additional inferences and references of their own and internalize the material, before it is re-visited by the teacher, so that students are in a better position to perform the higher-level inferences that are required for the higher stages of the model.

Context 2 provides further examples of teaching intertwined abstract concepts at different stages of concept formation. In this context, it can be seen that before moving up a stage of concept formation in the understanding (or application) of a certain skill, students may be required to jump to a lower stage of concept formation in the understanding or application of another skill. This reinforces our insight that the topics or even the skills developed using our framework need not be linear or prescribed in the order in which they should be taught. The modular nature of the framework indeed allowed for targeted reinforcement of abstraction skills at each stage, ensuring that students could integrate theoretical knowledge with practical problem-solving strategies. The reflective components of the framework provided valuable feedback loops, enabling continuous refinement of both teaching methods (and ensuring the teacher is aware of students' understanding) and student understanding itself.

6.2.3 How Educators Can Use the Framework. In reflecting on the application of our framework to the contexts, we noted a slight disconnect between what instructors are interested in and what the framework speaks of. This can be seen by the fact that all of the contexts describe teaching of a specific concept or topic. The vertical dimension of the framework (i.e., the stages of concept formation) are useful for thinking about how to teach a topic as they suggest a clear progression. The different abstraction skills, however, are not always apparent from the outset.

In some cases (e.g., when learning about a certain abstraction), there is an emphasis on the *identifying* skill since the goal is to learn about an abstraction or concept, with *moving* and *creating* appearing as "secondary skills" that are necessary and/or practiced simultaneously. In the case of concurrent programming, for example, the *moving* skill is required at *some* stages of learning a concept but not all.

In light of this, it is useful to identify two different ways in which educators can use the framework, summarising our insights from its application to our own contexts: (1) designing a module to teach something specific, and (2) analysing an existing module to identify opportunities for improvement.

When designing a new module or unit of teaching, one can start by considering the stages of the framework in order to plan a progression for learning some concept or abstraction. It is, of course, useful to try to map each activity to individual abstraction skills, and try to include as many stages of each skill as is suitable (e.g., if one realizes that U3 of *moving* is required, it is likely a good idea to cover U2 and U1 as well).

Another important consideration is that a skill such as *creating* may require students to first reach the threshold in the other two core skills first; this may particularly be the case at an advanced point in A1–A3. Conversely, at a concrete point in U1 and U2 *identifying* and *creating* may be very similar. These particularities are greatly dependent on the context, so we leave this up to the educator.

When analysing existing teaching, it is still interesting to look at the progression of the concept as a whole, but it is more important to map the individual abstraction skills that are used to the cognitive stages to ensure that the "higher" stages are preceded by corresponding "lower" stages to help students build their skills.

For both use cases, the educator does not need to fill in every step of the framework. However, a gap or leap in the stages of concept formation with respect to a skill or concept (e.g., beginning with U3 or leaping from A1 to A3) indicates a problem that should be addressed.

Finally, educators can use our framework in conjunction with other systematizations, which provide teaching tools to present concepts and foster skills at each stage. For example, Contexts 2 and 4 both make use of the Concreteness Fading [29] pedagogical framework. Additionally, because our framework is designed to be flexible and does not impose requirements around ordering of topics or content of curricula, it can also be applied within various approaches to CS education, ranging from traditional models to critical models (e.g., [54]).

This section has explored various examples of how the different contexts align with the framework. Tables 8 and 9 present a range of examples drawn from these contexts to offer a more practical resource for teachers. These tables show specific questions and activities that teachers can implement at each stage of the framework, providing actionable guidance to integrate the framework into their teaching practices.

7 Our Proposed Framework

In Section 4.1, we defined the following design criteria for our pedagogical framework. The framework should:

- use the conceptual stages of existing systemizations of abstraction as thresholds to describe how students develop understanding;
- guide students through these stages by using mental processes that require abstraction skills;
- be specific enough to be actionable within a Computer Science education context;
- be general enough that it is widely applicable within a Computer Science context.

Table 7: Our pedagogical framework showing the two outcome skills across the three stages of cognition & the three abstraction skills.

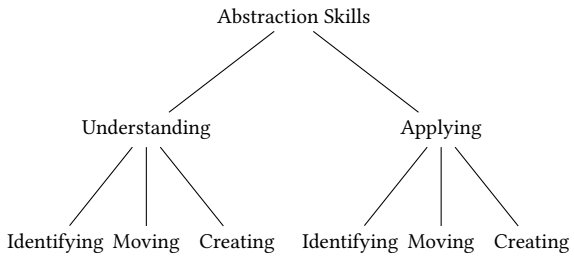
Outcome skill	Stage of cognition	Abstraction skill		
		Identifying	Moving	Creating
Understanding	U1: Observe	Seeing commonalities and differences between representations; finding an appropriate representation.	Switching between representations by adding or removing detail.	Constructing a representation. This can be from a single representation; or multiple (creating a new entity).
	U2: Induce and deduce			
	U3: Hypothesize and deduce			
Applying	A1: Observe			
	A2: Induce and deduce			
	A3: Hypothesize and deduce			

In its first iteration (Table 4), the framework includes three ordered stages of cognition (*observe*, *induce and deduce*, and *hypothesize and deduce*), which describe the conceptual stages of our framework and address our first design criterion. The framework identifies outcome skills (*understanding* and *applying*) and abstraction skills (*identifying*, *moving*, and *creating*), which offer guidance on helping students progress through the cognitive stages, meeting our second design criterion.

However, after applying the framework given in Table 4 to a range of teaching contexts (Section 5) and reflecting on its successes and limitations (Section 6), we felt we may be imposing an unnecessary structure on the framework, which in turn limits its applicability. Therefore, the framework needs adaptation in order to meet our third and fourth design criteria and enable application in a wide range of Computer Science education contexts.

As a result, we have revised the structure of our framework as described below and summarized in Table 7. Our proposed framework has three 'components' educators can use to help their students develop important skills or familiarize themselves with important concepts:

- Stages: these stages are hierarchical and allow the educator to think about the information the student has and what the educator expects them to do with it.
  - Stage 1: Observe
  - Stage 2: Induct and deduct
  - Stage 3: Hypothesize and deduct
- Abstraction skills:



  Understanding and applying are outcome skills, they must be considered as actions on a concept and the intricate skills must be considered in relation to abstraction levels.
- Tools: these are ways to aid in the design of prompts and questions that can help with the progression across a stage; this list is not necessarily exhaustive.
  - Block model
  - Fyfe's concreteness fading
  - Notional machines
  - Statter and Armoni's guidelines on how to use the PGK hierarchy in practice [64, 79].

This is summarized in Table 7. Tables 8 and 9 show good practices for the application of our framework taken from the case studies in Section 5.

The framework can be used to promote the development of 'outcome abstraction skills' of understanding and applying concepts; these skills are not teachable on their own, they must be developed in relation to a concept. Considering the other abstraction skills, identifying, moving and creating aid in helping this development whether teaching or learning. The framework can also be used to teach these three intricate skills, either in isolation or in conjunction with other intricate skills.

When used as a framework for the development of an outcome skill in relation to a given concept, the three other skills are mechanisms for helping the students progress through a stage and the end goal must indicate that the outcome skill has been acquired in relation to the concept.

When using the framework as a mechanism for fostering the intricate abstraction skills, the end goal must be some sort of indicator that the skill has been acquired.

7.1 Limitations

Our framework rests on other research, providing a synthesis of established models, frameworks, and pedagogy of abstraction skills from computer science and other fields. However, much like some of the systematizations that our framework is built on, our final work relies on certain assumptions.

For one, our understanding of abstraction skills is limited by our understanding of human behaviour and what this behaviour says about the cognitive process. The accuracy of empirical data

Table 8: Examples of scaffolding techniques for each stage and skill of the *understanding* outcome taken from the contexts in Section 5. The different techniques should be applicable to a large range of CS contexts.

Stage of cognition	Abstraction skill		
	Identifying	**Moving**	**Creating**
U1	* Use Concreteness Fading to identify key aspects (Context 2: Mock auction to identify players, strategies, and playoff. Context 4: Matryoshka dolls for recursion) * Identify problems that are unique to a concept/situation (Context 3: Problems unique to concurrent programming) * Illustrate new aspects of a concept/situation with code, live-coding, diagrams, software, ... (Context 3: Running code and observe different results each time Context 4: Algot as a visualization tool. Context 5: Presenting code).	* Use Concreteness Fading to move between concrete and abstract notions of a concept/situation (Context 2: Rock-Paper-Scissors) * Present diagrammatic representations (Context 5: Memory diagrams)	* Show a code and ask the students to modify it (Context 3: Synchronize a program) * Guide de students to observe specific properties (Context 5: Statements to observe the properties of a SSL) * Live coding while the students create code in parallel (Context 5: Add a new node at the end of a list)
U2	* Ask "what if" questions (Context 2: "What would have happened if ...")	* Use a notional machine to bridge from an existing model to another one (Context 1: Notional machine from existing memory diagram to block-and-arrow memory diagram) * Help students to generalize one or more solutions (Context 3: Consider the counter in the semaphore to be tracking a particular resource) * Ask students to trace code (Context 5: Add a new node to the end of a list)	* Having discussion on a code (Context 3: Importance of critical sections). * Ask students to explore alternative input values (Context 4: Recursive programs)
U3	* Ask "why" questions (Context 2: "Why did you choose ...") * Ask students to predict the output of a program (Context 3: The memory model)	* Discuss the differences between special cases of an instance (Context 3: Special case of a semaphore) *Ask students how shown problems generalize onto other inputs (Context 4: Recursive programs) * Give intermediate diagrammatic representations (Context 5: Intermediate memory diagrams)	* Highlight the difference between different approaches (Context 3: Creating abstractions) * Ask to create code from observing multiple representations (Context 5: Adding a new node to the beginning of the list) * Ask students to discuss their answers (Context 5: Adding a new node to the beginning of the list)

Table 9: Examples of scaffolding techniques for each stage and skill of the *applying* outcome taken from the contexts in Section 5. The different techniques should be applicable to a large range of CS contexts.

Stage of cognition	Abstraction skill		
	Identifying	**Moving**	**Creating**
A1	* Ask recall questions (Context 2: "How are payoffs calculated in an auction?")	* Show code and diagram representations of the same code (Context 1: Moving between abstractions of pointers)	* Ask students to explain code [modifications] (Context 5: Adding a new node to the beginning of the list)
A2	* Present problems where specific concepts are first explained in plain language using real-life examples (Context 4: Implementing recursive solutions "by recipe")	* Move from a representation to another one that is closer to the code implementation (Context 0: Recursion) * Move from the natural language description to an automatically generated code implementation using GenAI (e.g., "Prompt Problems" or "Explain in Plain English" exercises) (Section 2.4)	* Ask students to implement solutions using skeletons in which appropriate example input values have already been given (Context 4: Implementing recursive solutions "by recipe")
A3	* Ask students to identify the parts of the code that are working and are not working (Context 0: Recursion)	* Ask students to produce intermediate diagrammatic representations (Context 5: Removing from SSL)	* Apply prerequisite knowledge while teaching new concepts (Context 2: Apply prior understanding of payoffs to the new concept of auctions) * Give additional exercises (Context 3: Practice creating abstractions) * Ask students to generate test for edge cases and to test their solutions (Context 4: Test-driven approach)

is dependent on how, where and when it is collected; in turn, any insights obtained from the analysis of empirical data heavily depends on the soundness of the methodology. Our thematic analysis of the established systematizations was deductive; in other words our resulting framework contains the same biases as the theory it is grounded in, and our interpretation of the theory potentially adds to this bias.

Our evidence, being based in reflections from practitioners, is also subjective as it is limited by the context: content, competency of the cohort and mental model of the practitioner, amongst other things.

Our working group contained a broad range of expertise, which allowed us to cover a good range of the CS undergraduate curriculum through our chosen contexts. However, we acknowledge that there are other important contexts, such as artificial intelligence, that we could have addressed but did not. The selection of contexts reflects the expertise and focus areas of our group. We do not claim that the presented contexts form a complete representation of what CS undergraduates need to learn. Even if they did, the reflections would still be subject to the practitioner's bias. Therefore, we believe that as a tool for pedagogical reflection, our framework would benefit from insights contributed by a wider range of practitioners.

Finally, we cannot yet claim the framework improves student outcomes, this is something we feel we would like to explore in-situ; in turn, this evidence would lend some validity to our current and future insights.

8 Conclusion

By synthesizing existing pedagogical models of abstraction from Computer Science and adjacent fields we have constructed a novel framework for teaching abstraction. This synthesis in combination with the experience, reflections, and discussions of eleven Computer Science Education experts have resulted in the framework presented in this paper.

This framework has been demonstrated to possess the versatility necessary to be broadly applicable. From specific introductory concepts (e.g., pointer semantics in C) to broad more advanced topics (e.g., concurrent programming). In both cases we have demonstrated the framework's usefulness for evaluating and planning teaching activities to achieve desired learning outcomes for students. It is readily applicable to situations where the educator has practical constraints and cannot use tools that prescribe a linear order of teaching topics and/or skills, only prescribing that a student needs to progress through the three cognitive stages in sequence for a particular abstraction concept and abstraction skill.

In conclusion, we believe that the framework will be useful either as a separate tool or in combination with existing systematizations. Future empirical work remains to evaluate the effectiveness of the framework in the field and to provide additional insights into its applicability.

8.1 Future Work

Our current work can be extended by addressing some of the points in the limitations. In particular, controlled experimental studies based on the contexts could strengthen the validity argument of the paper. For example, the efficacy of our framework for teaching recursion in secondary school education (Context 4) can be tested in an empirical setting by applying the Basic Recursion Concept Inventory [36] as a posttest, either in its original form or in a modified version depending on the time available and the background of the students in a way that has been tested before [86].

Additional case studies from more diverse computer science learning contexts could also strengthen the framework by further demonstrating its versatility. With more use of the framework, some opportunities may arise for including additional subcategories of abstraction skills should that prove helpful.

Acknowledgments

We would like to thank Prof. Joseph Wood, Department of Computer Science, City St George's, University of London for his input in the discussions that led to the proposal for this working group.

This work was supported via grants and studentships by:

- the Doctoral College and Department of Computer Science at City and St Georges, University of London, United Kingdom.
- the graduate school CUGS at the Department of Computer and Information Science at Linköping University, Sweden;
- the National Recovery and Resilience Plan (NRRP), Mission 4, Component 2, Investment 1.1, Call for tender No. 104 published on 2/2/2022 by the Italian Ministry of University and Research (MUR), funded by the European Union – NextGenerationEU – Project Title "Learning Informatics" – CUP E53D23007720006 - Grant Assignment Decree No. 959 adopted on 22/04/2022 by the Italian Ministry of Ministry of University and Research (MUR).

References

[1] ACM Committee for Computing Education in Community Colleges (CCECC). 2023. *Bloom's for Computing: Enhancing Bloom's Revised Taxonomy with Verbs for Computing Disciplines.* Association for Computing Machinery, New York, NY, USA.

[2] Dan Aharoni. 2000. Cogito, Ergo sum! cognitive processes of students dealing with data structures. In *Proceedings of the Thirty-First SIGCSE Technical Symposium on Computer Science Education* (Austin, Texas, USA) *(SIGCSE '00)*. Association for Computing Machinery, New York, NY, USA, 26–30. https://doi.org/10.1145/330908.331804

[3] Shaaron Ainsworth. 2008. The Educational Value of Multiple-representations when Learning Complex Scientific Concepts. In *Visualization: Theory and Practice in Science Education*, John K. Gilbert, Miriam Reiner, and Mary Nakhleh (Eds.). Springer Netherlands, Dordrecht, 191–208. https://doi.org/10.1007/978-1-4020-5267-5_9

[4] John R. Anderson, Peter Pirolli, and Robert Farrell. 2014. Learning to program recursive functions. In *The nature of expertise*. Psychology Press, 153–183.

[5] Lorin W Anderson, David R Krathwohl, Peter W Airasian, Kathleen A Cruikshank, Richard E Mayer, Paul R Pintrich, James Raths, and Merlin C Wittrock. 2001. *A Taxonomy for Learning, Teaching, and Assessing: A Revision of Bloom's Taxonomy of Educational Objectives.* Pearson.

[6] Michal Armoni. 2013. On Teaching Abstraction in CS to Novices. *Journal of Computers in Mathematics and Science Teaching* 32, 3 (July 2013), 265–284. https://www.learntechlib.org/p/41271

[7] Alan C. Benander and Barbara A. Benander. 2008. Student monks–Teaching recursion in an IS or CS programming course using the Towers of Hanoi. *Journal of Information Systems Education* 19, 4 (2008), 455–467.

[8] John B. Biggs and Kevin F. Collis. 1982/2014. *Evaluating the Quality of Learning: The SOLO Taxonomy (Structure of the Observed Learning Outcome).* Academic Press. Google-Books-ID: xUO0BQAAQBAJ.

[9] Benjamin S. Bloom, Max D. Engelhart, Edward J. Furst, Walker H. Hill, David R. Krathwohl, et al. 1956. *Taxonomy of educational objectives: The classification of educational goals. Handbook 1: Cognitive domain.* Longman New York.

[10] Hans-J. Boehm and Sarita V. Adve. 2008. Foundations of the C++ Concurrency Memory Model. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Tucson, AZ, USA) *(PLDI '08)*. Association for Computing Machinery, New York, NY, USA, 68–78. https://doi.org/10.1145/1375581.1375591

[11] Axel Böttcher, Kathrin Schlierkamp, Veronika Thurner, and Daniela Zehetmeier. 2016. Teaching abstraction. In *2nd. International conference on higher education advances (HEAD'16)*. Editorial Universitat Politècnica de València, Editorial Universitat Politècnica de València, 357–364.

[12] Gillian M. Boulton-Lewis. 1995. The SOLO Taxonomy as a Means of Shaping and Assessing Learning in Higher Education. *Higher Education Research & Development* 14, 2 (1995), 143–154. https://doi.org/10.1080/0729436950140201

[13] Jerome S. Bruner. 1966. *Toward a Theory of Instruction.* Belknap Press of Harvard University, Cambridge, MA.

[14] Francisco Enrique Vicente Castro and Kathi Fisler. 2017. Designing a multi-faceted SOLO taxonomy to track program design skills through an entire course. In *Proceedings of the 17th Koli Calling International Conference on Computing Education Research (Koli Calling '17)*. Association for Computing Machinery, New York, NY, USA, 10–19. https://doi.org/10.1145/3141880.3141891

[15] Ibrahim Cetin and Ed Dubinsky. 2017. Reflective abstraction in computational thinking. *The Journal of Mathematical Behavior* 47 (2017), 70–80. https://doi.org/10.1016/j.jmathb.2017.06.004

[16] College Board. 2021. *AP Computer Science A: Course Overview* (updated january 2021 ed.). College Board. https://apcentral.collegeboard.org/media/pdf/ap-computer-science-a-course-overview.pdf

[17] Cornelia Connolly, Eamonn Murphy, and Sarah Moore. 2006. Introducing the APOS Model to Teaching Computing at Higher Education. In *Proceedings of the 9th International Conference on Engineering Education.* iNEER, 4–11. https://www.ineer.org/Events/ICEE2006/papers/3260.pdf

[18] Julia Crossley. 2023. How do Students Conceptualize and Represent Abstract Ideas? An Initial Exploration.. In *Proceedings of the 2023 ACM Conference on International Computing Education Research - Volume 2* (Chicago, IL, USA) *(ICER '23)*. Association for Computing Machinery, New York, NY, USA, 82–86. https://doi.org/10.1145/3568812.3603455

[19] Julia Crossley. 2023. Processes of Abstraction and Representation: An Initial Exploration.. In *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 2* (Turku, Finland) *(ITiCSE 2023)*. Association for Computing Machinery, New York, NY, USA, 617–618. https://doi.org/10.1145/3587103.3594148

[20] Julia Crossley. 2024. Exploring a framework of computational expression. In *Proceedings of the 2024 UK and Ireland Computing Education Research conference (UKICER '24)*. ACM, New York, NY, USA, 3 pages.

[21] Julia Crossley, Marjahan Begum, and Joseph Wood. 2023. Qualitative insights into abstractions skills from exam scripts. In *Proceedings of the European Computer Summit.* Informatics Europe.

[22] Quintin Cutts, Sarah Esper, Marlena Fecho, Stephen R. Foster, and Beth Simon. 2012. The abstraction transition taxonomy: developing desired learning outcomes through the lens of situated cognition. In *Proceedings of the Ninth Annual International Conference on International Computing Education Research* (Auckland, New Zealand) *(ICER '12)*. Association for Computing Machinery, New York, NY, USA, 63–70. https://doi.org/10.1145/2361276.2361290

[23] Paul Denny, Juho Leinonen, James Prather, Andrew Luxton-Reilly, Thezyrie Amarouche, Brett A. Becker, and Brent N. Reeves. 2024. Prompt Problems: A New Programming Exercise for the Generative AI Era. In *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1* (Portland, OR, USA) *(SIGCSE 2024)*. Association for Computing Machinery, New York, NY, USA, 296–302. https://doi.org/10.1145/3626252.3630909

[24] Paul Denny, James Prather, Brett A. Becker, James Finnie-Ansley, Arto Hellas, Juho Leinonen, Andrew Luxton-Reilly, Brent N. Reeves, Eddie Antonio Santos, and Sami Sarsa. 2024. Computing Education in the Era of Generative AI. *Commun. ACM* 67, 2 (Jan. 2024), 56–67. https://doi.org/10.1145/3624720

[25] Avinash Dixit. 2005. Restoring fun to game theory. *The Journal of Economic Education* 36, 3 (2005), 205–219.

[26] E. Dubinski. 1991. Reflective Abstraction in Advanced Mathematical Thinking. *Advanced Mathematical Thinking* (1991).

[27] Ursula Fuller, Colin G. Johnson, Tuukka Ahoniemi, Diana Cukierman, Isidoro Hernán-Losada, Jana Jackova, Essi Lahtinen, Tracy L. Lewis, Donna McGee Thompson, Charles Riedesel, and Errol Thompson. 2007. Developing a computer science-specific learning taxonomy. *SIGCSE Bull.* 39, 4 (Dec. 2007), 152–170. https://doi.org/10.1145/1345375.1345438

[28] Emily R Fyfe, Nicole M McNeil, Ji Y Son, and Robert L Goldstone. 2014. Concreteness fading in mathematics and science instruction: A systematic review. *Educational psychology review* 26 (2014), 9–25.

[29] Emily R. Fyfe and Mitchell J. Nathan. 2019. Making "concreteness fading" more concrete as a theory of instruction for promoting transfer. *Educational Review*

71, 4 (2019), 403–422. https://doi.org/10.1080/00131911.2018.1424116
[30] Aakash Gautam, Whitney Bortz, and Deborah Tatar. 2020. Abstraction Through Multiple Representations in an Integrated Computational Thinking Environment. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education* (Portland, OR, USA) *(SIGCSE '20)*. Association for Computing Machinery, New York, NY, USA, 393–399. https://doi.org/10.1145/3328778.3366892
[31] Maximilian Georg Barth, Sverrir Thorgeirsson, and Zhendong Su. 2024. A Direct Manipulation Programming Environment for Teaching Introductory and Advanced Software Testing. In *Proceedings of the 24th Koli Calling International Conference on Computing Education Research (Koli Calling '24)*. Association for Computing Machinery, New York, NY, USA, Article 2, 11 pages. https://doi.org/10.1145/3699538.3699564
[32] Carlisle E George. 2000. EROSI—visualising recursion and discovering new errors. *ACM SIGCSE Bulletin* 32, 1 (2000), 305–309.
[33] David Ginat and Yoav Blau. 2017. Multiple Levels of Abstraction in Algorithmic Problem Solving. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education* (Seattle, Washington, USA) *(SIGCSE '17)*. Association for Computing Machinery, New York, NY, USA, 237–242. https://doi.org/10.1145/3017680.3017801
[34] David Ginat and Eti Menashe. 2015. SOLO Taxonomy for Assessing Novices' Algorithmic Design. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education (SIGCSE '15)*. Association for Computing Machinery, New York, NY, USA, 452–457. https://doi.org/10.1145/2676723.2677311
[35] Oliver Graf, Sverrir Thorgeirsson, and Zhendong Su. 2024. Assessing Live Programming for Program Comprehension. In *Proceedings of the 2024 on Innovation and Technology in Computer Science Education V. 1* (Milan, Italy) *(ITiCSE 2024)*. Association for Computing Machinery, New York, NY, USA, 520–526. https://doi.org/10.1145/3649217.3653547
[36] Sally Hamouda, Stephen H Edwards, Hicham G Elmongui, Jeremy V Ernst, and Clifford A Shaffer. 2017. A basic recursion concept inventory. *Computer Science Education* 27, 2 (2017), 121–148.
[37] Orit Hazzan. 1999. Reducing Abstraction Level When Learning Abstract Algebra Concepts. *Educational Studies in Mathematics* 40 (1999), 71–90. Issue 1.
[38] Orit Hazzan. 2002. Reducing abstraction level when learning computability theory concepts. In *Proceedings of the 7th Annual Conference on Innovation and Technology in Computer Science Education* (Aarhus, Denmark) *(ITiCSE '02)*. Association for Computing Machinery, New York, NY, USA, 156–160. https://doi.org/10.1145/544414.544461
[39] Orit Hazzan. 2008. Reflections on teaching abstraction and other soft ideas. *SIGCSE Bull.* 40, 2 (June 2008), 40–43. https://doi.org/10.1145/1383602.1383631
[40] Orit Hazzan and Jeff Kramer. 2016. Assessing abstraction skills. *Commun. ACM* 59, 12 (dec 2016), 43–45. https://doi.org/10.1145/2926712
[41] Jonathan H. Hill, Bernice J. Houle, Susan M. Merritt, and Allen Stix. 2008. Applying abstraction to master complexity. In *Proceedings of the 2nd International Workshop on The Role of Abstraction in Software Engineering* (Leipzig, Germany) *(ROA '08)*. Association for Computing Machinery, New York, NY, USA, 15–21. https://doi.org/10.1145/1370164.1370169
[42] Charles A. Holt and Monica Capra. 2000. Classroom games: A prisoner's dilemma. *The Journal of Economic Education* 31, 3 (2000), 229–236.
[43] Cruz Izu. 2022. Modelling the Use of Abstraction in Algorithmic Problem Solving. In *Proceedings of the 27th ACM Conference on on Innovation and Technology in Computer Science Education Vol. 1* (Dublin, Ireland) *(ITiCSE '22)*. Association for Computing Machinery, New York, NY, USA, 193–199. https://doi.org/10.1145/3502718.3524758
[44] Cruz Izu, Amali Weerasinghe, and Cheryl Pope. 2016. A Study of Code Design Skills in Novice Programmers using the SOLO taxonomy. In *Proceedings of the 2016 ACM Conference on International Computing Education Research (ICER '16)*. Association for Computing Machinery, New York, NY, USA, 251–259. https://doi.org/10.1145/2960310.2960324
[45] Joint Task Force on Computing Curricula, Association for Computing Machinery (ACM) and IEEE Computer Society. 2020. *Computing Curricula 2020: Paradigms for Global Computing Education*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3467967
[46] Hieke Keuning, Isaac Alpizar-Chacon, Ioanna Lykourentzou, Lauren Beehler, Christian Köppe, Imke de Jong, and Sergey Sosnovsky. 2024. Students' Perceptions and Use of Generative AI Tools for Programming Across Different Computing Courses. In *Proceedings of the 24th Koli Calling International Conference on Computing Education Research (Koli Calling '24)*. Association for Computing Machinery, New York, NY, USA, Article 14, 12 pages. https://doi.org/10.1145/3699538.3699546
[47] Yifat Ben-David Kolikant. 2004. Learning Concurrency as an Entry Point to the Community of Computer Science Practitioners. *Journal of Computers in Mathematics and Science Teaching* 23, 1 (2004), 21–46.
[48] Jeff Kramer. 2007. Is abstraction the key to computing? *Commun. ACM* 50, 4 (apr 2007), 36–42. https://doi.org/10.1145/1232743.1232745
[49] Leslie Lamport. 1979. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Trans. Comput.* C-28, 9 (1979), 690–691. https://doi.org/10.1109/TC.1979.1675439
[50] Elynn Lee, Victoria Shan, Bradley Beth, and Calvin Lin. 2014. A structured approach to teaching recursion using cargo-bot. In *Proceedings of the Tenth Annual Conference on International Computing Education Research* (Glasgow, Scotland, United Kingdom) *(ICER '14)*. Association for Computing Machinery, New York, NY, USA, 59–66. https://doi.org/10.1145/2632320.2632356
[51] Dalit Levy. 2001. Insights and conflicts in discussing recursion: A case study. *Computer Science Education* 11, 4 (2001), 305–322. https://doi.org/10.1076/csed.11.4.305.3829
[52] Michael Xieyang Liu, Advait Sarkar, Carina Negreanu, Benjamin Zorn, Jack Williams, Neil Toronto, and Andrew D. Gordon. 2023. "What It Wants Me To Say": Bridging the Abstraction Gap Between End-User Programmers and Code-Generating Large Language Models. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems* (Hamburg, Germany) *(CHI '23)*. Association for Computing Machinery, New York, NY, USA, Article 598, 31 pages. https://doi.org/10.1145/3544548.3580817
[53] Violetta Lonati, Dario Malchiodi, Mattia Monga, and Anna Morpurgo. 2017. Nothing to Fear but Fear Itself: Introducing Recursion in Lower Secondary Schools. In *2017 International Conference on Learning and Teaching in Computing and Engineering (LaTICE)*. 91–98. https://doi.org/10.1109/LaTiCE.2017.23
[54] James W. Malazita and Korryn Resetar. 2019. Infrastructures of abstraction: how computer science education produces anti-political subjects. *Digital Creativity* 30, 4 (2019), 300–312. https://doi.org/10.1080/14626268.2019.1682616 arXiv:https://doi.org/10.1080/14626268.2019.1682616
[55] Susana Masapanta-Carrión and J. Ángel Velázquez-Iturbide. 2018. A Systematic Review of the Use of Bloom's Taxonomy in Computer Science Education. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education (SIGCSE '18)*. Association for Computing Machinery, New York, NY, USA, 441–446. https://doi.org/10.1145/3159450.3159491
[56] Claudio Mirolo, Cruz Izu, Violetta Lonati, and Emanuele Scapin. 2022. Abstraction in Computer Science Education: An Overview. *Informatics in Education* 20, 4 (2022), 615–639. https://doi.org/10.15388/infedu.2021.27
[57] Orna Muller and Bruria Haberman. 2008. Supporting abstraction processes in problem solving through pattern-oriented instruction. *Computer Science Education* 18, 3 (2008), 187–212. https://doi.org/10.1080/08993400802332548
[58] Laurie Murphy, Renée McCauley, and Sue Fitzgerald. 2012. 'Explain in plain English' questions: implications for teaching. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education* (Raleigh, North Carolina, USA) *(SIGCSE '12)*. Association for Computing Machinery, New York, NY, USA, 385–390. https://doi.org/10.1145/2157136.2157249
[59] Liat Nakar, Mor Friebroon, and Michal Armoni. 2024. From Modelling to Assessing Algorithmic Abstraction – the Missing Dimension. In *Proceedings of the 23rd Koli Calling International Conference on Computing Education Research* (Koli, Finland) *(Koli Calling '23)*. Association for Computing Machinery, New York, NY, USA, Article 4, 12 pages. https://doi.org/10.1145/3631802.3631815
[60] Thomas L. Naps, Guido Rößling, Vicki Almstrum, Wanda Dann, Rudolf Fleischer, Chris Hundhausen, Ari Korhonen, Lauri Malmi, Myles McNally, Susan Rodger, and J. Ángel Velázquez-Iturbide. 2002. Exploring the role of visualization and engagement in computer science education. In *Working Group Reports from ITiCSE on Innovation and Technology in Computer Science Education* (Aarhus, Denmark) *(ITiCSE-WGR '02)*. Association for Computing Machinery, New York, NY, USA, 131–152. https://doi.org/10.1145/960568.782998
[61] Greg L. Nelson, Filip Strömbäck, Ari Korhonen, Marjahan Begum, Ben Blamey, Karen H. Jin, Violetta Lonati, Bonnie MacKellar, and Mattia Monga. 2020. Differentiated Assessments for Advanced Courses that Reveal Issues with Prerequisite Skills: A Design Investigation. In *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education* (Trondheim, Norway) *(ITiCSE-WGR '20)*. Association for Computing Machinery, New York, NY, USA, 75–129. https://doi.org/10.1145/3437800.3439204
[62] Keiron Nicholson, Judith Good, and Katy Howland. 2009. Concrete Thoughts on Abstraction. In *Proceedings of the 21th Annual Workshop of the Psychology of Programming Interest Group*. 8. https://ppig.org/files/2009-PPIG-21st-nicholson.pdf
[63] Jacob Perrenet, Jan Friso Groote, and Eric Kaasenbrood. 2005. Exploring students' understanding of the concept of algorithm: levels of abstraction. *SIGCSE Bull.* 37, 3 (jun 2005), 64–68. https://doi.org/10.1145/1151954.1067467
[64] Jacob Perrenet and Eric Kaasenbrood. 2006. Levels of abstraction in students' understanding of the concept of algorithm: the qualitative perspective. *SIGCSE Bull.* 38, 3 (June 2006), 270–274. https://doi.org/10.1145/1140123.1140196
[65] Jean Piaget. 1972. *The Principles of Genetic Epistemology*. Routledge and Kegan Paul, London, UK.
[66] James Prather, Paul Denny, Juho Leinonen, Brett A. Becker, Ibrahim Albluwi, Michelle Craig, Hieke Keuning, Natalie Kiesler, Tobias Kohn, Andrew Luxton-Reilly, Stephen MacNeil, Andrew Petersen, Raymond Pettit, Brent N. Reeves, and Jaromir Savelka. 2023. The Robots Are Here: Navigating the Generative AI Revolution in Computing Education. In *Proceedings of the 2023 Working Group Reports on Innovation and Technology in Computer Science Education* (Turku, Finland) *(ITiCSE-WGR '23)*. Association for Computing Machinery, New York, NY, USA, 108–159. https://doi.org/10.1145/3623762.3633499

[67] James Prather, Brent N Reeves, Juho Leinonen, Stephen MacNeil, Arisoa S Randrianasolo, Brett A. Becker, Bailey Kimmel, Jared Wright, and Ben Briggs. 2024. The Widening Gap: The Benefits and Harms of Generative AI for Novice Programmers. In *Proceedings of the 2024 ACM Conference on International Computing Education Research - Volume 1* (Melbourne, VIC, Australia) *(ICER '24)*. Association for Computing Machinery, New York, NY, USA, 469–486. https://doi.org/10.1145/3632620.3671116

[68] Inioluwa Deborah Raji, Morgan Klaus Scheuerman, and Razvan Amironesei. 2021. You Can't Sit With Us: Exclusionary Pedagogy in AI Ethics Education. In *Proceedings of the 2021 ACM Conference on Fairness, Accountability, and Transparency* (Virtual Event, Canada) *(FAccT '21)*. Association for Computing Machinery, New York, NY, USA, 515–525. https://doi.org/10.1145/3442188.3445914

[69] Ian Sanders and Tamarisk Scholtz. 2012. First year students' understanding of the flow of control in recursive algorithms. *African Journal of Research in Mathematics, Science and Technology Education* 16, 3 (2012), 348–362.

[70] Eddie Antonio Santos and Brett A. Becker. 2024. Not the Silver Bullet: LLM-enhanced Programming Error Messages are Ineffective in Practice. In *Proceedings of the 2024 Conference on United Kingdom & Ireland Computing Education Research* (Manchester, United Kingdom) *(UKICER '24)*. Association for Computing Machinery, New York, NY, USA, Article 5, 7 pages. https://doi.org/10.1145/3689535.3689554

[71] Carsten Schulte. 2008. Block Model: an educational model of program comprehension as a tool for a scholarly approach to teaching. In *Proceedings of the Fourth International Workshop on Computing Education Research* (Sydney, Australia) *(ICER '08)*. Association for Computing Machinery, New York, NY, USA, 149–160. https://doi.org/10.1145/1404520.1404535

[72] Andrew D. Selbst, Danah Boyd, Sorelle A. Friedler, Suresh Venkatasubramanian, and Janet Vertesi. 2019. Fairness and Abstraction in Sociotechnical Systems. In *Proceedings of the Conference on Fairness, Accountability, and Transparency* (Atlanta, GA, USA) *(FAT* '19)*. Association for Computing Machinery, New York, NY, USA, 59–68. https://doi.org/10.1145/3287560.3287598

[73] Ana Selvaraj, Eda Zhang, Leo Porter, and Adalbert Gerald Soosai Raj. 2021. Live Coding: A Review of the Literature. In *Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education V. 1* (Virtual Event, Germany) *(ITiCSE '21)*. Association for Computing Machinery, New York, NY, USA, 164–170. https://doi.org/10.1145/3430665.3456382

[74] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. 2010. X86-TSO: A Rigorous and Usable Programmer's Model for X86 Multiprocessors. *Commun. ACM* 53, 7 (July 2010), 89–97. https://doi.org/10.1145/1785414.1785443

[75] Anna Sfard. 1991. On the dual nature of mathematical conceptions: Reflections on processes and objects as different sides of the same coin. *Educational Studies in Mathematics* 22, 1 (Feb. 1991), 1–36. https://doi.org/10.1007/bf00302715

[76] Richard R. Skemp. 1976. Relational Understanding and Instrumental Understanding. *Mathematics Teaching* 77 (1976), 20–26.

[77] David H. Smith, Paul Denny, and Max Fowler. 2024. Prompting for Comprehension: Exploring the Intersection of Explain in Plain English Questions and Prompt Writing. In *Proceedings of the Eleventh ACM Conference on Learning @ Scale* (Atlanta, GA, USA) *(L@S '24)*. Association for Computing Machinery, New York, NY, USA, 39–50. https://doi.org/10.1145/3657604.3662039

[78] Juha Sorva. 2013. Notional Machines and Introductory Programming Education. *ACM Transactions on Computing Education* 13 (06 2013), 8:1–8:31. https://doi.org/10.1145/2483710.2483713

[79] David Statter and Michal Armoni. 2020. Teaching Abstraction in Computer Science to 7th Grade Students. *ACM Trans. Comput. Educ.* 20, 1, Article 8 (jan 2020), 37 pages. https://doi.org/10.1145/3372143

[80] Filip Strömbäck, Linda Mannila, Mikael Asplund, and Mariam Kamkar. 2019. A Student's View of Concurrency - A Study of Common Mistakes in Introductory Courses on Concurrency. In *Proceedings of the 2019 ACM Conference on International Computing Education Research* (Toronto ON, Canada) *(ICER '19)*. ACM, New York, NY, USA, 229–237. https://doi.org/10.1145/3291279.3339415

[81] Filip Strömbäck, Linda Mannila, and Mariam Kamkar. 2021. The Non-Deterministic Path to Concurrency – Exploring how Students Understand the Abstractions of Concurrency. *Informatics in Education* 20, 4 (2021), 683–715. https://doi.org/10.15388/infedu.2021.29

[82] Filip Strömbäck, Linda Mannila, and Mariam Kamkar. 2022. A Weak Memory Model in Progvis: Verification and Improved Accuracy of Visualizations of Concurrent Programs to Aid Student Learning. In *Koli Calling '22: 22nd Koli Calling International Conference on Computing Education Research* (Koli, Finland) *(Koli 2022)*. Association for Computing Machinery, New York, NY, USA, Article 14, 12 pages. https://doi.org/10.1145/3564721.3565947

[83] Keiichi Takaya. 2008. Jerome Bruner's Theory of Education: From Early Bruner to Later Bruner. *Interchange* 39 (2008), 1–19. https://doi.org/10.1007/s10780-008-9039-2

[84] David Tall and Michael Thomas. 2002. *Intelligence, learning and understanding in mathematics.* Post Pressed, Flaxton, QLD, Australia.

[85] Josh Tenenberg. 2019. Qualitative Methods for Computing Education. In *The Cambridge Handbook of Computing Education Research*, Sally A. Fincher and Anthony V. Robins (Eds.). Cambridge University Press, 173–207. https://doi.org/10.1017/9781108654555.008

[86] Sverrir Thorgeirsson, Lennart C. Lais, Theo B. Weidmann, and Zhendong Su. 2024. Recursion in Secondary Computer Science Education: A Comparative Study of Visual Programming Approaches. In *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1* (Portland, OR, USA) *(SIGCSE 2024)*. Association for Computing Machinery, New York, NY, USA, 1321–1327. https://doi.org/10.1145/3626252.3630916

[87] Sverrir Thorgeirsson and Zhendong Su. 2021. Algot: An Educational Programming Language with Human-Intuitive Visual Syntax. In *2021 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 1–5. https://doi.org/10.1109/VL/HCC51201.2021.9576166

[88] Sverrir Thorgeirsson, Theo B. Weidmann, Karl-Heinz Weidmann, and Zhendong Su. 2024. Comparing Cognitive Load Among Undergraduate Students Programming in Python and the Visual Language Algot. In *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1* (Portland, OR, USA) *(SIGCSE 2024)*. Association for Computing Machinery, New York, NY, USA, 1328–1334. https://doi.org/10.1145/3626252.3630808

[89] Sverrir Thorgeirsson, Chengyu Zhang, Theo B. Weidmann, Karl-Heinz Weidmann, and Zhendong Su. 2024. An Electroencephalography Study on Cognitive Load in Visual and Textual Programming. In *Proceedings of the 2024 ACM Conference on International Computing Education Research - Volume 1* (Melbourne, VIC, Australia) *(ICER '24)*. Association for Computing Machinery, New York, NY, USA, 280–292. https://doi.org/10.1145/3632620.3671124

[90] Annapurna Vadaparty, Daniel Zingaro, David H. Smith IV, Mounika Padala, Christine Alvarado, Jamie Gorson Benario, and Leo Porter. 2024. CS1-LLM: Integrating LLMs into CS1 Instruction. In *Proceedings of the 2024 on Innovation and Technology in Computer Science Education V. 1* (Milan, Italy) *(ITiCSE 2024)*. Association for Computing Machinery, New York, NY, USA, 297–303. https://doi.org/10.1145/3649217.3653584

[91] J. Ángel Velázquez-Iturbide. 2021. An Analysis of the Formal Properties of Bloom's Taxonomy and Its Implications for Computing Education. In *Proceedings of the 21st Koli Calling International Conference on Computing Education Research* (Joensuu, Finland) *(Koli Calling '21)*. Association for Computing Machinery, New York, NY, USA, Article 17, 7 pages. https://doi.org/10.1145/3488042.3488069

[92] Juan Diego Tascón Vidarte, Christian Rinderknecht, Jee-In Kim, and HyungSeok Kim. 2010. A Tangible Interface for Learning Recursion and Functional Programming. In *2010 International Symposium on Ubiquitous Virtual Reality*. IEEE, 32–35. https://doi.org/10.1109/ISUVR.2010.18

[93] Theo B. Weidmann, Sverrir Thorgeirsson, and Zhendong Su. 2022. Bridging the Syntax-Semantics Gap of Programming. In *Proceedings of the 2022 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (Auckland, New Zealand) *(Onward! 2022)*. Association for Computing Machinery, New York, NY, USA, 80–94. https://doi.org/10.1145/3563835.3567668

[94] Uri Wilensky. 1991. Abstract meditations on the concrete and concrete implications for mathematics education. In *Constructionism*, Idit Harel and Seymour Papert (Eds.). Ablex Publishing, 193–203.

[95] Claes Wohlin. 2014. Guidelines for snowballing in systematic literature studies and a replication in software engineering. In *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering* (London, England, United Kingdom) *(EASE '14)*. Association for Computing Machinery, New York, NY, USA, Article 38, 10 pages. https://doi.org/10.1145/2601248.2601268

[96] Chunpeng Zhai, Santoso Wibowo, and Lily D. Li. 2024. The effects of over-reliance on AI dialogue systems on students' cognitive abilities: a systematic review. *Smart Learning Environments* 11, 1 (2024), 37 pages. https://doi.org/10.1186/s40561-024-00316-7

A Threading Library in C

Below is the implementation of the primitives used in the concurrency examples in the paper. The names and semantics are the same as in the educational operating system Pintos. The code below is an implementation of the same semantics using Pthreads. This makes it possible to compile the same program both inside of Pintos as well as on other Posix-based systems (e.g., Linux). Therefore, students don't need to learn two different set of names initially. The code below is typically not shown to students when teaching the concepts, as it is just glue-code. Teaching rather focuses on the implementation inside Pintos. Since that implementation is exclusively in kernel mode, it does not have to account for system calls etc. and is therefore easier to understand compared to what would exist in e.g. Linux or other operating systems.

```
#pragma once

#ifndef _POSIX_C_SOURCE
#define _POSIX_C_SOURCE 20000101L
#endif

#include <stdbool.h>
#include <pthread.h>
#include <semaphore.h>
#include <time.h>
#include <errno.h>

struct semaphore {
  sem_t os;
};

void sema_init(struct semaphore *sema, unsigned
    value);
void sema_destroy(struct semaphore *sema);
void sema_down(struct semaphore *sema);
void sema_up(struct semaphore *sema);

struct lock {
  pthread_mutex_t os;
};

void lock_init(struct lock *lock);
void lock_destroy(struct lock *lock);
void lock_acquire(struct lock *lock);
void lock_release(struct lock *lock);

typedef void thread0_func(void);
typedef void thread1_func(void *aux);
void thread_new(thread0_func *fn);
void thread_new1(thread1_func *fn, void *aux);
```

Listing 27: Threading library, header file

```
#include "os.h"
#include <stdlib.h>

struct thread_start {
  thread_func1 *run;
  void *aux;
  struct semaphore started;
};

static void *thread_main(struct thread_start
    *start) {
  thread_func1 *run = start->run;
  void *aux = start->aux;

  sema_up(&start->started);

  (*run)(aux);

  return NULL;
}

void thread_new1(thread1_func *fn, void *aux) {
  struct thread_start info;
  info.run = fn;
  info.aux = aux;
  sema_init(&info.started, 0);

  pthread_t created;
  if (pthread_create(&created, NULL,
    &thread_main, &info)) {
    perror("pthread_create");
    exit(1);
  }

  sema_down(&info.started);
  sema_destroy(&info.started);

  // Same semantics as in Pintos:
  pthread_detach(created);
}

static void start0(void *aux) {
  thread0_func *fn = aux;
  (*fn)();
}

void thread_new(thread0_func *fn) {
  thread_new1(&start0, fn);
}


void sema_init(struct semaphore *sema, unsigned
    value) {
    sem_init(&sema->os, 0, value);
}

void sema_destroy(struct semaphore *sema) {
    sem_destroy(&sema->os);
}

void sema_down(struct semaphore *sema) {
    while (sem_wait(&sema->os) != 0) {
```

```
        if (errno != EINTR) {
            perror("sem_wait");
            exit(1);
        }
    }
}

void sema_up(struct semaphore *sema) {
    sem_post(&sema->os);
}

/**
 * Lock.
 */

void lock_init(struct lock *lock) {
    pthread_mutexattr_t attr;
    pthread_mutexattr_init(&attr);
    pthread_mutexattr_settype(&attr,
    PTHREAD_MUTEX_ERRORCHECK);
    pthread_mutex_init(&lock->os, &attr);
    pthread_mutexattr_destroy(&attr);
}

void lock_destroy(struct lock *lock) {
    pthread_mutex_destroy(&lock->os);
}

void lock_acquire(struct lock *lock) {
    pthread_mutex_lock(&lock->os);
}

void lock_release(struct lock *lock) {
    pthread_mutex_unlock(&lock->os);
}
```

Listing 28: Threading library, implementation file