A Domain Analysis of Data Structure and Algorithm Explanations in the Wild

Jeffrey Young
Oregon State University
School of EECS
Corvallis, Oregon, USA

Eric Walkingshaw
Oregon State University
School of EECS
Corvallis, Oregon, USA

ABSTRACT

Explanations of data structures and algorithms are complex interactions of several notations, including natural language, mathematics, pseudocode, and diagrams. Currently, such explanations are created ad hoc using a variety of tools and the resulting artifacts are static, reducing explanatory value. We envision a domain-specific language for developing rich, interactive explanations of data structures and algorithms. In this paper, we analyze this domain to sketch requirements for our language. We perform a grounded theory analysis to generate a qualitative coding system for explanation artifacts collected online. This coding system implies a common structure among explanations of algorithms and data structures. We believe this structure can be reused as the semantic basis of a domain-specific language for creating interactive explanation artifacts. This work is part of our effort to develop the paradigm of explanation-oriented programming, which shifts the focus of programming from computing results to producing rich explanations of how those results were computed.

CCS CONCEPTS

• **Social and professional topics** → **Computing education**;

KEYWORDS

explanation-oriented programming; domain-specific languages; grounded theory; algorithm explanation

ACM Reference Format:
Jeffrey Young and Eric Walkingshaw. 2018. A Domain Analysis of Data Structure and Algorithm Explanations in the Wild. In *Proceedings of The 49th ACM Technical Symposium on Computer Science Education (SIGCSE'18).* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3159450.3159477

1 INTRODUCTION

Data structures and algorithms are at the heart of computer science and must be explained to each new generation of students. How can we do this effectively? In this paper, we focus on the *artifacts* that constitute or support explanations of data structures and algorithms (hereafter just "algorithms"), which can be shared and reused. For verbal explanations, such as a lecture, the supporting artifact might be the associated slides. For written explanations, the artifact is the explanation as a whole, including the text and any supporting figures. Explanation artifacts for algorithms are interesting because they typically present a complex interaction among many different notations, including natural language, mathematics, pseudocode, executable code, various kinds of diagrams, animations, and more.

Currently, explanation artifacts for algorithms are created ad hoc using a variety of tools and techniques, and the resulting explanations tend to be static, reducing their explanatory value. Although there has been a substantial amount of work on algorithm visualization [8, 9, 11–13, 19], and tools exist for creating these kinds of supporting artifacts, there is no good solution for creating integrated, multi-notational explanations as a whole. Similarly, although some algorithm visualization tools provide a means for the student to tweak the parameters or inputs to an algorithm to generate new visualizations, they do not support creating cohesive interactive explanations that correspondingly modify the surrounding explanation or allow the student to respond to or query the explanation in other ways. To fill this gap, we envision a *domain-specific language* (DSL) that supports the creation of rich, interactive, multi-notational artifacts for explaining algorithms. The development of this DSL is part of an effort to explore the new paradigm of *explanation-oriented programming*, described in Section 2.1.

The intended users of the envisioned DSL are CS educators who want to create *interactive artifacts* to support the explanation of algorithms. These users are experts on the corresponding algorithms and also skilled programmers. The produced explanation artifacts might supplement a lecture or be posted to a web page as a self-contained (textual and graphical) explanation. The DSL should support pedagogical methods through built-in abstractions and language constructs. It should also support a variety of forms of student interaction. For example, teachers should be able to define equivalence relations enabling users to automatically generate variant explanations [6], to build in responses to anticipated questions, and to provide explanations at multiple levels of abstraction.

This paper presents a formative step toward this vision. We conduct a *qualitative analysis* of our domain to determine the form and content of the explanation artifacts that educators are already creating. Specifically, we answer the following research questions:

RQ1. What are the component parts of an algorithm explanation?
RQ2. How does each part advance the overall explanation?
RQ3. How are the parts of an algorithm explanation structured?
RQ4. What kinds of notations are used in algorithm explanations?

Answers to these questions set expressiveness requirements for our DSL since we should be able to express explanations that educators are already creating. They also guide the design of the DSL by providing an initial set of components and operations for building explanations. We base our analysis on the established qualitative research method of *grounded theory* [20], described in Section 2.2.

SIGCSE'18, February 21–24, 2018, Baltimore, MD, USA

For our analysis, we collected 15 explanation artifacts from the internet, as described in Section 3. These artifacts are lecture notes that explain two algorithms and one data structure commonly covered in undergraduate computer science courses: Dijkstra's shortest path algorithm [14, pp. 137–142], merge sort [14, pp. 210–214], and AVL trees [15, pp. 458–475]. Applying grounded theory, we develop a coding system that captures the components, structure, and notations of the explanation in each document. This paper makes the following contributions:

C1. We provide a *coding system* (Section 4.2) for analyzing explanation artifacts in the form of lecture notes. We show that through the application of the coding system, each artifact, regardless of content, author, or institution, forms a tree structure, which we have termed an *explanation tree* (Section 4.3).
C2. We provide a coded qualitative data set of explanation artifacts, using the system defined in C1, applied to our sample of 15 collected explanation artifacts, along with a tool for exploring and visualizing this data set (Section 4.1).
C3. We describe how our coding system and explanation trees can provide a semantic basis for a DSL and argue for the advantages of such an approach (Section 5).

2 BACKGROUND AND RELATED WORK

In this section, we put our work into context. In Section 2.1, we describe explanation-oriented programming, which motivates our work. In Section 2.2, we describe the grounded theory methodology that we used to develop our coding system. And in Section 2.3, we briefly discuss other work related to teaching algorithms.

2.1 Explanation-Oriented Programming

Explanation-oriented programming (XOP) is a programming paradigm where the primary output of a program is not a set of computed values, but an *explanation of how* those values were computed [3–6, 21]. A high-level goal of this work is to further realize the paradigm of XOP through the development of a specific DSL.

Programming languages for XOP should not merely produce explanations as a byproduct but should provide abstractions and features specific to the creation of explanation artifacts. For example, they should provide a way to use application-specific notations and visualizations (which are widespread in explanations of algorithms), and to define or generate alternative explanations in response to user input [6]. Additionally, languages for XOP should help guide the programmer toward the creation of *good* explanations.

The need for interactive explanation artifacts is motivated by the observation that there is a trade-off between personal explanations and traditional explanation artifacts, which can be partially bridged by XOP programs viewed as rich, interactive explanation artifacts. A good *personal explanation* is useful because the explainer can *respond* to the student, adjusting the pace and strategy as necessary. For example, the teacher can answer questions, rephrase parts of an explanation, and provide additional examples as needed. Unfortunately, good personal explanations are a scarce resource. First, there are limited number of people who can provide high quality personal explanations on a topic. Second, a personal explanation is usually ephemeral and so cannot be directly shared or reused. Since personal explanations are hard to come by, many students learn from *impersonal explanation artifacts*, such as recorded lectures, textbooks, and online written and graphical resources. These impersonal explanations lack the interaction and adaptability of personal explanations, but have the advantage of being easy to massively share and reuse via libraries and the internet.

In-person lectures exist at a midway point between impersonal and personal explanations, perhaps closer to the personal end of the spectrum. These *classroom explanations* are adaptable—students can ask questions in class, the teacher can respond, and explanations can be adapted on the fly if students are confused—but less so than personal explanations since the teacher must accommodate many students at once. Classroom explanations are shared among many students, but less shareable than impersonal explanations since they are ephemeral and therefore difficult to reuse.

We target another midway point, closer to the impersonal end of the spectrum, of *interactive explanation artifacts* that attempt to reproduce the responsiveness and adaptability of personal explanations, but which can still be shared and reused online. Such explanation artifacts would be expensive to produce with current tools since an *explanation designer* must not only create a high quality initial explanation and corresponding visualizations, but also anticipate and explicitly program responses to queries by the student. We expect that DSLs for XOP can help alleviate this burden.

2.2 Grounded Theory

The core idea of grounded theory is to generate or discover a theory inductively, based on data, rather than using data to evaluate a theory developed a priori. Grounded theory is rooted in a pragmatist view that theory should target its intended uses [20]. Our study uses the grounded theory methodology defined by Corbin et al. [2].

Grounded theory starts by collecting initial data on the subject of interest. For example, a researcher interested in why students drop out of computer science programs might conduct interviews with students, and collect student schedules and homework assignments. Once some data is collected, the researcher begins *coding*, which is the process of assigning descriptive tags to qualitative data.

Coding consists of three stages: (1) During *open coding*, the researcher writes down *any* terms that describe the data. (2) During *axial coding*, the researcher identifies similarities and other relationships between tags developed during open coding. The goal of this step is to develop a *coding paradigm*, which is a model that describes the inter-relationship of tags. (3) Finally, during *selective coding*, the researcher identifies a small set of core tags that capture the main concepts and relationships identified during axial coding. These tags form the basis of the theory extracted from the data.

In grounded theory, data collection and analysis occurs simultaneously and iteratively. That is, after forming an initial theory, new data will be added that might provide new tags during open coding, which will suggest revisions to the theory developed through axial and selective coding, which will trigger a re-analysis of old data, and so on. This back-and-forth movement between data collection, analysis, and theory building is a marked departure from quantitative methods where phases are distinct [20].

How does the researcher know when the theory is adequate and no new data is required? There are three tenets that can help answer that question, which are pivotal to the validity of the grounded

theory method [20]. (1) The tenet of *constant comparison* is that during all phases of coding, the researcher must constantly return to earlier data to check whether tags are applied consistently. (2) The tenet of *theoretical sampling* focuses on filling perceived gaps in the data based on the current theory. For example, we chose to analyze explanations of algorithms operating on three different underlying data types: lists, trees, and graphs. After analyzing explanations of merge sort, explanations of other sorting algorithms have less theoretical sampling value than explanations of a very different kind of algorithm. (3) The tenet of *saturation* helps determine when to stop collecting and coding new data. Saturation occurs when the coding system is able to accommodate new data without modification. That is, when the theory can accurately describe data that was *not* used to generate it. The amount of data needed to reach this point will vary depending on the topic, research questions, and individual researchers.

2.3 Teaching Algorithms through Artifacts

Our underlying motivation is to create artifacts that help explain algorithms. This motivation is shared by a long tradition of researchers working on algorithm visualization [8–13, 17, 19]. This work is complementary to our goal since a visualization might be one part of a comprehensive, multi-notational explanation of an algorithm. Although our formative study considers only static explanations in the form of lecture notes, our ultimate goal is to design a DSL that enables the creation of interactive explanation artifacts that realize some of the benefits of personal explanations. Others have demonstrated that interactivity makes video lectures a more effective pedagogical tool [16, 18, 22].

3 EXPERIMENTAL SETUP

In this section we describe how we executed the grounded theory process outlined in Section 2.2. We restricted the scope of data collection to include only lecture notes produced by faculty/instructors in computer science departments at respected universities. Furthermore, we restrict our data to include only static, written documents. Restricting the scope in this way has two benefits: (1) All explanatory artifacts share an intrinsic goal to communicate the mechanics, application, or implementation of a common computer science algorithm. (2) There are many and varied examples of different approaches to explain the same algorithm, and many examples of similar approaches to explain different algorithms. All data collected was either in PDF format, or in HTML format and converted to PDF. We considered only self-contained lecture notes, excluding slides which we view as incomplete explanations without the associated presentation they support. We made no attempt to restrict the size of the data collected. The smallest document collected was 1.5 pages and the longest was 18 pages.

All data was collected and coded by hand by the first author with the aid of Atlast.ti software.[1] Atlast.ti provides direct support for the grounded theory process by allowing open coding; easing constant comparison with operations for organizing, searching, and filtering coded documents; and easing axial and selective coding with operations for collecting, merging, and revising extent tags.

We focused data collection on three algorithms based on different underlying data types: Dijkstra's shortest path algorithm on graphs [14, pp. 137–142], merge sort of lists [14, 210–214], and the AVL tree implementation of balanced binary search trees [15, pp. 458–475]. We achieved saturation during the grounded theory analysis after 11 lecture notes. As additional validation and to round out our data set, we continued collecting and coding documents until our sample included 5 lecture notes on each of the 3 topics, for 15 total explanation artifacts.

Some documents contain explanations of multiple algorithms. In cases where secondary algorithms are explained as part of the explanation of the target algorithm (e.g. as necessary background), we coded the explanation as usual. In cases where multiple algorithms are explained in a single document but the explanations are unconnected, we did not code the others. For example, one document explaining Dijkstra's algorithm also provided an explanation of Bellman-Ford's shortest-path algorithm, which we ignored.

4 RESULTS

In this section we present the results of our grounded theory analysis. In Section 4.1, we briefly describe the coded data set and a simple associated tool that we provide. In Section 4.2 we describe the coding system produced by the grounded theory analysis and provide a sample coding to illustrate its use. We observe that the coding paradigm identified during axial coding is that explanations are tree structured. In Section 4.3, we discuss the tree structure of explanations and describe how to transform a coded explanation artifact into the corresponding explanation tree.

4.1 Data Set and Tool Support

The coded data set is available online.[2] The coded documents are provided both in the proprietary Atlast.ti format and in an exported CSV format. Artifacts are indexed according to the algorithm of focus and a simple counter. For example, the first AVL tree document is named AVT01 and the fifth is named AVT05.

We also provide a small tool written in Haskell that implements the coding system described in Section 4.2 and the explanation tree representation described in Section 4.3. The tool supports converting a code sequence into the corresponding explanation tree and rendering explanation trees in a 2-dimensional plain-text format (see Figure 3). We provide the code sequence for each document in the data set in a format compatible with the tool.

4.2 The Coding System

The coding system consists of four finite sets of tags—aspects, moves, roles, and notations—identified by the grounded theory analysis. Aspects and moves are summarized in Table 1, roles and notations in Table 2.

An *aspect* tag identifies a constituent part of an algorithm explanation, that is, *what* is being discussed (answering RQ1). Example aspects of an explanation are the goal of the algorithm, required operations, historical context, advantages, disadvantages, and implementation details. A *move* tag is always associated with a parent aspect and describes *how* that aspect is addressed (RQ2), that is,

[1] http://atlasti.com/

[2] https://github.com/lambda-land/XOP-Algorithms-Data

Aspect	
Advantage	pro or upside
Algorithm	a specific algorithm
Application	a use case or application
Class	a group, set, or class
Complexity	computational complexity
Constituent	a constituent part of the parent aspect
Data Structure	a specific data structure
Design	design considerations
Disadvantage	con or downside
Goal	a goal or desired outcome
History	historical background
Implementation	implementation details
Motivation	motivation or rationale
Operation	a specific operation
Problem	a problem to be solved
Property	a condition, invariant, or property
Solution	a solution to a prior problem
State	a modifiable state
Move	
Abstraction	generalize or abstract from the aspect
Assumption	highlight introduce an assumption
Cases	break something down into cases
Comment	a dummy move for orphaned decorators
Conclusion	wrap-up or reflect
Contrast	contrast this aspect with another
Definition	define a concept
Derivation	derive through a sequence of steps
Description	give a general description
Example	provide an example
Implication	logical implication, if … then …
InVivo	define a term that practitioners use
Legend	provide a key to help interpret a diagram
Observation	offer a general observation
Outline	provide a point-by-point overview
Proof	provide a formal proof
Proposal	suggest a path forward
Solicitation	ask something of the reader
Summary	summarize preceding aspects

Table 1: Overview of all *aspects* and *moves* identified by the grounded theory analysis. Aspects organize an explanation into its constituent parts while moves are the specific steps taken to guide the reader toward understanding.

the step taken by the explainer to help advance the reader's understanding. For example, by breaking the aspect into cases, providing an example or proof, or defining a concept. A single aspect may be explained in several moves. We take the name for this concept from Bellack et al. [1]'s notion of a "pedagogical move".

Both aspect and move tags can be decorated by tags describing secondary roles and notations. A *role* tag modifies a move or aspect to indicate that this part of the explanation serves some secondary purpose not directly captured by the modified tag (related to both RQ2 and RQ3). For example, attaching an Aside role to a move might indicate that the move is an aside that does not directly advance the explanation of the parent aspect, while the Pedagogical role might indicate that the move gives advice about how to study an aspect rather than explaining it. A *notation* tag modifies a move or aspect to indicate that this part of the explanation is presented in some format other than natural language text (RQ4). If no notation decorator is provided, the tagged fragment is assumed to be text.

Role	
Aside	not directly related to parent aspect
Caveat	clarifies or qualifies a point
Meta	about this aspect/move rather than advancing it
Pedagogical	addresses a pedagogical concern
Related	substantially related to another aspect/move
Review	info that is implied to have already been covered
Notation	
Cartoon	a drawn or animated graphic
Code	a block of code in a programming language
Mathematics	mathematical functions, formulas, or equations
PseudoCode	non-executable code-like language
Sequence	an ordered, bulleted, or punctuated list
Table	infromation in tabular format

Table 2: Overview of the secondary *roles* and *notations* identified by the grounded theory analysis. These codes decorate aspect and move codes to add additional information.

$$
\begin{array}{rcl}
a \in \mathit{Aspect} \quad m \in \mathit{Move} \quad r \in \mathit{Role} \quad n \in \mathit{Notation} & & \\
c \in \mathit{Code} & ::= & \Rightarrow a \mid \Leftarrow \mid m \mid c + d \\
d \in \mathit{Decorator} & ::= & r \mid n
\end{array}
$$

Figure 1: Syntax of codes from grounded theory analysis.

During axial coding, we observed that the meaning of an individual code often depends on preceding codes. For example, a description move does not stand on its own but typically describes a preceding aspect. Understanding how a move advances an explanation may require understanding a sequence of preceding aspects, but not necessarily the immediately preceding ones. For example, in AVT02 we observe the sequence *data structure, problem, solution, description, property, definition* where the final move defines a property of the data structure; the intervening subsequence *problem, solution, description* is irrelevant to understanding the definition but forms a separate dependency chain. Thus, it seems explanations have a hierarchical structure (RQ3) that the coding system must capture. To do this, we introduce structuring elements to our codes to indicate where in the hierarchy a given aspect or move sits.

The syntax of codes is defined by the grammar in Figure 1, which refers to the tags defined in Tables 1 and 2. A new child aspect a is introduced by a "push" code, $\Rightarrow a$, and we exit out of this aspect with a corresponding "pop" code, $\Leftarrow$. A sibling aspect can be added to the hierarchy with a pop followed by a push. Since adding siblings is quite common, we introduce $\Leftrightarrow a$ as syntactic sugar for this case. A move tag m is added as a child to the current aspect, which may not be the most recent aspect named in a code due to intervening pop codes. Finally, secondary roles and notations are unified as *decorators*; a decorator d can be added to a code c as $c + d$.

In Table 3, we illustrate the application of the coding system to the beginning of one of the documents in our data set, MS03. The document begins in Row 1 with a header that we code as the

	Text	Coding
1	*2.2 Mergesort*	⇒ Algorithm
2	The algorithms that we consider in this section is based on a simple operation known as merging: combining two ordered arrays to make one larger ordered array.	⇒ Operation Description InVivo
3	This operation immediately lends itself to a simple recursive sort method known as mergesort: to sort an array, divide it into two halves, sort the two halves (recursively), and then merge the results.	⇐ Definition + Sequence
4	⟨diagram of list⟩	+ Cartoon
5	Mergesort guarantees to sort an array of N items in time proportional to N log N, no matter what the input.	⇒ Motivation Description + Mathematics
6	Its prime disadvantage is that it uses extra space proportional to N.	⇔ Disadvantage Description + Mathematics

Table 3: Sample text and codings for beginning of MS03. Italicized text corresponds to headers in the original document.

root aspect, representing the algorithm itself. Row 2 addresses a sub-aspect, the merge operation, which it explains in two moves: a description, and the definition of an in-vivo term. Row 3 is coded by a pop since the explanation is no longer focused on this operation, but is defining what merge sort is. This move is decorated by the sequence notation since it is defined by a sequence of steps. Row 4, referring to a diagram illustrating this definition, further decorates this move with the cartoon notation. Row 5 concerns the sub-aspect of motivating merge sort, which it does through a description move that uses (light) mathematical notation. Row 6 introduces the sibling aspect of disadvantages of merge sort, which is also explained through a description with mathematical notation.

4.3 Explanation Trees

In the previous section, we described how our analysis revealed a hierarchical structure that is crucial to understanding how an explanation does its work. We call this hierarchical structure an *explanation tree*. The internal nodes of an explanation tree are the hierarchy of aspects that an explanation addresses, while the leaves of the tree are the specific moves taken to explain these aspects. Both the aspects and moves of an explanation tree can be decorated by secondary roles and notations. The notion of an explanation tree, discovered by considering the relationship between adjacent moves during axial coding, became the *coding paradigm* for our analysis, so we adapted our codes to capture this structure.

Our coding system can be viewed as a simple stack-based language for constructing an explanation tree. Figure 2 defines an interpretation of codes as operations in a stack-based machine that builds an explanation tree. The semantic domain of codes under this interpretation is an update function on a stack of trees, represented as a linked list. The $\Rightarrow\ a$ code pushes a node labeled by a with no children to the stack. Note that we use a to represent both the aspect and the corresponding tree node on the stack. The $\Leftarrow$ code pops the first two nodes on the stack and adds the first node as a child of the second. The m code adds the move node m to the stack and then immediately executes a pop code ($\Leftarrow$) since moves correspond to leaves in the explanation tree. In this interpretation, we assume that the codes have been preprocessed to incorporate all decorators into the corresponding nodes. Using the $[\![\cdot]\!]$ function we can build an explanation tree from a sequence of codes by simply left-folding the function over the codes with an initially empty stack, then executing pop codes until the stack is empty.

$$
\begin{aligned}
[\![\cdot]\!] &: \mathit{Tree}^* \to \mathit{Tree}^* \\
[\![\Rightarrow a]\!](ts) &= a :: ts \\
[\![\Leftarrow]\!](c :: p :: ts) &= \mathit{addChild}(c, p) :: ts \\
[\![m]\!](ts) &= [\![\Leftarrow]\!](m :: ts)
\end{aligned}
$$

Figure 2: Interpreting codes as operations in a stack machine that builds an explanation tree.

```
Aspect Algorithm
+- Aspect Operation
|  +- Move Description
|  `- Move InVivo
+- Move Definition @ [Note Sequence,Note Cartoon]
+- Aspect Motivation
|  `- Move Description @ [Note Mathematics]
`- Aspect Disadvantage
   `- Move Description @ [Note Mathematics]
```

Figure 3: Tree rendering of the codes for MS03 in Table 3.

The Haskell tool provided with the data set implements the transformation from code sequences to explanation trees. It also provides a way to render explanation trees. This functionality is illustrated in Figure 3, which renders the explanation tree produced by the code sequence in the coding sample in Table 3.

A move at the leaf of an explanation tree can be understood to advance the reader's understanding of the aspects along the path to a root. For example, the last description move explains a disadvantage of the algorithm identified by the root. The linear structure of the original explanation can be recovered by a pre-order traversal of the explanation tree.

5 DISCUSSION

This formative study was conducted to better understand the domain of algorithm and data structure explanations, in order to inform the design of an explanation-oriented DSL for creating interactive explanation artifacts. In this section, we interpret the results from Section 4 in this context.

At a basic level, the grounded theory analysis reveals a set of concepts that should be realized by abstractions, constructs, and features of the DSL. The DSL should provide ways to capture the various aspects of an algorithm explanation, including aspects like historical background, motivation, and advantages/disadvantages that are typically not captured formally. The DSL should also support a range of more and less formal mechanisms for advancing an explanation through pedagogical moves. The DSL should provide formal mechanisms for (semi-)automatically generating examples from implementations, performing case-analyses and derivations, and testing properties of an algorithm, but also support informal

explanatory moves such as presenting observations and assumptions. The DSL should also provide support for a variety of different notations and support secondary roles such as asides and caveats.

Explanation trees can serve as an underlying semantic model for the DSL. In an interactive setting, explanation trees provide convenient places (nodes) to hang extra details about an aspect that may be explored or not. Lecture notes enforce one path through an explanation, but interactive explanations can provide multiple paths through the same tree structure corresponding to explanations at different levels of abstraction and with different focuses. In addition to providing suggested paths through the tree, we can allow users to navigate the tree on their own, exploring more details or alternative explanations where desired. The secondary roles we discovered are evidence that such supplementary information that does not directly advance the explanation is useful. However, there is a disincentive in a linear explanation to provide too many extra details that detract from the main line of the explanation. These concerns would be mitigated in an interactive setting where secondary details can be explored on-demand and remain out-of-the-way otherwise.

A more speculative application is to use our theory to adapt and remix explanations into new ones targeted at a specific audience or even an individual user. By organizing an explanation into its constituent aspects that describe *what* part of an explanation is doing, and separating this from the moves that describe *how* it is doing it, it is possible to identify other ways of explaining the same thing. This empirical approach to generating alternative explanations can supplement other strategies explored in previous work, such as generating alternative explanations from equivalence laws [6].

Finally, it is important to acknowledge some limitations of this work. First, our results are the output of a grounded theory analysis which is inherently subjective. Grounded theory provides a framework for systematically extracting a theory from qualitative data (see Section 2.2), but the process relies on our subjective decisions about how to code the data. It is possible that other researchers follow the same process and yet arrive at very different results.

Another limitation is that our decision to sample only lecture notes excludes many other ways that people already explain algorithms, such as oral or animated explanations. We restricted our study to lecture notes because they were easy to collect and analyze and because they represent coherent, self-contained explanations, which is the focus of our proposed DSL. However, unlike animations or algorithm visualization tools, lecture notes are static. Since we aim to produce interactive explanation artifacts, it is possible we missed important features because of this restriction.

Finally, since explanations are about communicating ideas, it is possible that *discourse analysis* [7], which focuses on extracting meaning from *how language is used*, would have been a better choice than grounded theory. We chose grounded theory because it is a process we are familiar with and because we anticipated coding artifacts with a variety of notations besides natural language.

6 CONCLUSION

This paper presented a grounded theory analysis of 15 explanations of algorithms and data structures in the form of lecture notes. The analysis yielded a coding paradigm that organizes algorithm explanations into explanation trees, where internal nodes represent aspects of the algorithm and leaves represent pedagogical moves that incrementally advance the reader's understanding. We make our coded data set publicly available and provide an associated tool for rendering the explanation tree corresponding to each document.

This study was performed as a formative domain analysis to inform the design of a DSL for creating interactive explanations of algorithms and data structures. Explanation trees can provide a semantic basis for such a language, and that the enumerated aspects, moves, and roles discovered by the analysis suggest a suite of useful constructs and abstractions the language should support and establishes its basic expressiveness requirements.

ACKNOWLEDGMENTS

Thanks to Shujin Wu for helping collect data and conceptualize this work. Thanks to Dan Hillman for providing data from a previous study, which we didn't end up using but very much appreciate.

REFERENCES

[1] A. A. Bellack, F. L. Smith, H. M. Kliebard, and R. T. Hyman. 1966. *The Language of the Classroom.* Teachers College Press.
[2] J. Corbin, A. Strauss, and A. L. Strauss. 2014. *Basics of Qualitative Research.* SAGE.
[3] M. Erwig and E. Walkingshaw. 2008. A Visual Language for Representing and Explaining Strategies in Game Theory. In *IEEE Int. Symp. on Visual Languages and Human-Centric Computing (VL/HCC).* 101–108.
[4] M. Erwig and E. Walkingshaw. 2009. A DSL for Explaining Probabilistic Reasoning. In *IFIP Working Conf. on Domain-Specific Languages (DSL) (LNCS)*, Vol. 5658. 335–359.
[5] M. Erwig and E. Walkingshaw. 2009. Visual Explanations of Probabilistic Reasoning. In *IEEE Int. Symp. on Visual Languages and Human-Centric Computing (VL/HCC).* 23–27.
[6] M. Erwig and E. Walkingshaw. 2013. A Visual Language for Explaining Probabilistic Reasoning. *Journal of Visual Languages and Computing (JVLC)* 24, 2 (2013), 88–109.
[7] J. P. Gee. 2014. *An Introduction to Discourse Analysis: Theory and Method* (4 ed.). Routledge.
[8] P. Gloor. 1997. Animated Algorithms. In *Elements of Hypermedia Design: Techniques for Navigation & Visualization in Cyberspace.* Birkhäuser, Boston, 235–241.
[9] P. A. Gloor. 1992. AACE – Algorithm Animation for Computer Science Education. In *IEEE Workshop on Visual Languages.* 25–31.
[10] S. Grissom, M. F. McNally, and T. Naps. 2003. Algorithm Visualization in CS Education: Comparing Levels of Student Engagement. In *ACM Symp. on Software Visualization (SoftVis).* 87–94.
[11] S. Hansen, N. H. Narayanan, and M. Hegarty. 2002. Designing Educationally Effective Algorithm Visualizations. *Journal of Visual Languages and Computing (JVLC)* 13, 3 (2002), 291–317.
[12] C. D. Hundhausen, S. A. Douglas, and J. T. Stasko. 2002. A Meta-Study of Algorithm Visualization Effectiveness. *Journal of Visual Languages and Computing (JVLC)* 13 (2002), 259–290.
[13] C. Kann, R. W. Lindeman, and R. Heller. 1997. Integrating algorithm animation into a learning environment. *Computers and Education (CE)* 28, 4 (1997), 223–228.
[14] J. Kleinberg and E. Tardos. 2006. *Algorithm design.* Pearson Education, Boston.
[15] D. E. Knuth. 1998. *The Art of Computer Programming: Sorting and Searching* (2 ed.). Pearson Education, Boston.
[16] M. Merkt, S. Weigand, A. Heier, and S. Schwan. 2011. Learning with Videos vs. Learning with Print: The Role of Interactive Features. *Learning and Instruction (LI)* 21, 6 (2011), 687–704.
[17] T. L. Naps, G. Rößling, V. Almstrum, W. Dann, R. Fleischer, C. Hundhausen, A. Korhonen, L. Malmi, M. McNally, S. Rodger, and J. A. Velàzquez-Iturbide. 2002. Exploring the Role of Visualization and Engagement in Computer Science Education. *SIGCSE Bulletin* 35, 2 (June 2002), 131–152.
[18] S. Schwan and R. Riempp. 2004. The Cognitive Benefits of Interactive Videos: Learning to Tie Nautical Knots. *Learning and Instruction (LI)* 14, 3 (2004), 293–305.
[19] C. A. Shaffer, M. L. Cooper, A. J. D. Alon, M. Akbar, M. Stewart, S. Ponce, and S. H. Edwards. 2010. Algorithm Visualization: The State of the Field. *ACM Transactions on Computing Education (TOCE)* 10, 3 (2010), 9.
[20] A. Strauss and J. Corbin. 1967. Discovery of Grounded Theory. (1967).
[21] E. Walkingshaw and M. Erwig. 2011. A DSEL for Studying and Explaining Causation. In *IFIP Working Conf. on Domain-Specific Languages (DSL).* 143–167.
[22] D. Zhang. 2005. Interactive Multimedia-Based e-Learning: A Study of Effectiveness. *The American Journal of Distance Education (AJDE)* 19, 3 (2005), 149–162.