Programming paradigms for computational science: three fundamental models

Miguel-Angel Sicilia, Elena García-Barriocanal, Salvador Sánchez-Alonso, and Marçal Mora-Cantallops

Computer Science Department, University of Alcalá,
Polytechnic Building. Ctra. Barcelona km. 33.6
28871 Alcalá de Henares (Madrid), Spain
{msicilia, elena.garciab, salvador.sanchez, marcal.mora}@uah.es

Abstract. The widespread of data science programming languages and libraries have raised new interest in teaching computational science coding in ways that leverage the capabilities of both single-computer and cluster-based computation infrastructures. Some of the programming patterns and idioms are converging, yet there are specialized uses and cases that require learners to switch from one to another. In this paper, we report on the experience in action research with more than ten cohorts of mixed background students in postgraduate level data science classes. We first discuss the key mental models found to be essential to understanding solution design, and then review the three fundamental paradigms that students must face when coding data manipulation and their interrelation. Finally, we discuss some insights on additional elements found important in understanding the specificities of current practice in data analysis tasks.

Keywords: Computational science · Education · Data science · Programming · Mental models.

1 Introduction

Computational science requires programming skills that are to a large extent determined by the languages, libraries, frameworks and applications used for the various domains of scientific inquiry. These are not only related to the creation of models themselves but to the broader scope of data manipulation, which have been captured in the past as part of knowledge discovery and data mining process frameworks [13]. The success of the concept of a *data scientist* [9][5] as a professional role that deals with data-intensive problems and has a broad and hybrid skill set has to some extent predated the idea of a computational scientist. Arguably, data scientists that are in the domain of some particular scientific discipline are computational scientists. This idea is reflected in the EDISON framework for data science education [16], that includes a competence area that incorporates scientific methods closer to experimental work as it is done in the sciences than to applied data analytics in a business context.

ICCS Camera Ready Version 2019
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-22750-0_33

In any case, teaching programming for computational science or data science is a different endeavor than teaching general-purpose programming as it is practiced in undergraduate degrees in the domain of computing. Typically, data science study programs are offered to audiences with a diverse profile [6], and it is expected in them that students with a background different from computing are able to effectively develop software that makes use of existing scientific libraries and computing infrastructures, witouth them becoming software engineers. Typical languages taught in data science include R, Python (using SciPy libraries) and Julia, and less frequently nowadays, Octave, Matlab, SAS or others. Some of them as Python are actually general-purpose languages that were not originally devised with data anlysis as a goal. However, data wrangling and libraries consuming data (as machine learning libraries) in them tend to follow some particular idioms or paradigms that need not be the same as those used in the language. An example is the difference of using NumPy arrays [21] and Pandas dataframes [17] in Python, which is widely different to using regular Python lists, both in logical and in internal representation aspects. There are elements of efficiency and memory handling, typing and even style that make the experience of data analysis in Python very different from programming Python for other purposes, e.g. to develop a Web site.

Here we describe insights from the experience in teaching scientific programming to non-computer science graduate students. Concretely, it reports on the reflections of the experience in ten cohorts of students enrolled in programs related to data science (namely: Data Science, Business Intelligence and Bussiness Analytics and Big Data) that were taught programming for data science in Python (and secondarily classic statistical analysis programming with R) at the University of Alcalá in Madrid in the last five years. The results come from the observation of common pitfalls and difficulties found when approaching the grading of assignments, and from subsequent interviews focused on particular problems in understanding. Incremental inquiry in cycles was done applying action research principles, similar to those that have been applied to teaching programming elsewhere [15]. We have come up with a number of mental models and programming styles or paradigms that require separate attention and are different in relevant aspects. Further, we report on a number of additional elements that were found important in an adequate understanding of the specifics of programming in data science.

The rest of this paper is structured as follows. Section 2 describes the assumptions, background and overall setting of the educational programs that have served as the basis for the discussion. Then, in Section 3 we deal with the key mental models identified that students have to develop in order to efficiently code in data science settings. Section 4 describes the three fundamental programming paradigms or models identified that require significant cognitive effort when moving from one to the other. In Section 5, we discuss additional elements that have been found as important in providing students the adequate context to understand the tasks at hand. Finally, conclusions and outlook are provided in Section 6.

2 Background and assumptions

We assume here that teaching data science can be considered as a broader concept than that of teaching computational science, as the latter focuses on models for the diverse scientific disciplines, while the former is more inclusive of computational models not necesarily related to the enterprise of science. Actually, there are some experiences reported on explicitly combining both concepts, e.g. Giabbanelli et al. [11] report on a course combining computational models with data science concepts. In other cases, both concepts are considered as overlapping when discussing educational aspects, e.g. in [8]. It should also be noted that data tasks as the application of machine learning are a current active area in online question-answering sites [1] reflecting the importance of the topic.

The typical path for teaching data science starts with exposing the students to introductory lessons on programming, typically using a high level language as Python (as in [6] for example). This entails the usual sequence of introducing first variables, conditional and iterative control structures, then some fundamental data structures: lists and dictionaries. However, data science environments adhere to a paradigm sometimes called *array-oriented programming* in which operations on scalars apply transparently to vectors, matrices, and higher-dimensional arrays. This results in a degree of conciseness, as operations abstract out the number of dimensions of the data, and for most common operations, it is rare the need to use control structures. Instead, vectorized operations are applied on the data, and there are facilities to select and transform data with operators. Examples are boolean indexing in arrays or clauses that specialize in selecting subsets of data and generalize the notion of slicing lists. In consequence, the teaching of basic Python used in our approach does not emphsaize algorithm design, but the use of a core of data structures and the application of functions. Particularly, object oriented design (encapsulation, inheritance, etc.) is not included in the teaching, and the understanding of object orientation is limited to the syntax for sending messages to objects (i.e. invoking methods), which is natural in languages as Python.

We assume here that the teaching is directed to a non-computer science student, so that the emphasis is on using libraries and not on developing new algorithms, parallel versions of them or optimized versions of some complex user interfaces, which would require other kind of skills. Also, we assume that database or data store access will be made transparent using some form of SQL-like languages. It should be noted that these assumptions are rather strong and somewhat controversial. For example, in the case of tasks as data acquisition from APIs (application programmer interfaces as those exposed in RESTful interfaces), data wrangling would require some extra abilities in some cases. In any case, in this paper we focus on the core of the daily activities of a data scientist that involve array and dataframe manipulation and application of library functions, even though there is currently a degree of ambiguity on data science team roles and the skills required for different profiles [18].

3 Key mental models

Mental models are concise abstract representations that can help shape behaviour and set an approach to solving problems. It has been recognized that mental models are required by novices to learn programming [7]. The assumption of mental models is that we construct mental “small-scale models” of reality, which can be used to anticipate events, to reason, and to underlie explanation. These models are simplified and incomplete, but are helpful in our construction of knowledge on a topic.

Most students attending data science classes already have a mental model consisting of “table like structures” similar to those used in spreadsheets. Many of them had some knowledge of the relational model and the SQL language so that the ideas of tables is natural. However, there are other mental models that have been found to be essential in our experience and that require dedicated attention in educational programs, which are discussed in what follows.

3.1 Array programming: vectorization and broadcasting

Array programming revolves around the idea of having multidimensional arrays, and operations are applied over those irrespective of their size or number of dimensions. The key concept is that of *vectorized* operations. Library functions for example in SciPy are already vectorized, so that they can be applied to arrays of any number of dimensions. Also, it is possible to automatically vectorize functions. The following example is from SciPy, but similar alternatives are available in other languages as R.

```
def _transform(x,y):
    if (x>0.5*y and y<0.3):
        return (sin(x-y))
    elif (x<0.5*y):
        return 0
    elif (x>0.2*y):
        return (2*sin(x+2*y))
    else:
        return (sin(y+x))
transform = np.vectorize(_transform, otypes=[np.float])
transform(a, b) // a, b can be of any number of dimensions
```

It should be noted that such vectorization does in general not provide any performance advantage over using `for` loop iteration. However, vectorized operations in libraries are typically optimized explicitly or implemented in lower level languages as C or Fortran, so that they should always be preferred.

Broadcasting is an effect related to applying operations on arrays of different sizes. Under some constraints, one of the arrays is broadcast, meaning that it is repeated so that the sizes are compatible. This occurs trivially in expressions as `a*5` where `a` is an array, and the scalar is broadcast to the same size of the array.

The understanding of vectorization and broadcasting is critical to all operations, as even the selection of elements is based on vectorized logical or relational operators. This sets repetitive control structures (looping) as an exception that is largely not needed for manipulating arrays and matrices (even though it is still required for other tasks, as processing data from APIs or Web pages, but that can be considered as different tasks).

The mental model is then that of functions that can be used with arrays of any dimensions replacing looping, including implicit extension on dimensions. This requires a change in the mental model of students in the case they have experience in non-vectorized programming and tend to recur to loops as a default strategy.

3.2 Memory hierarchy, paralelism and task models

The second important mental model requires basic knowledge of computer architecture, in an abstract way. The fundamental model is that of a hierarchy of memory, typically considering as levels: (a) internal registers in the processor, (b) system RAM and (c) external storage as in disks. This nowadays gets an additional level in that of (d) distributed storage, e.g. in a cluster of computers. The latter is dealt here considering that the programming constructs are transparent to the actual parallel computing infrastructure, instead of dealing with concrete models as in the case of MPI reported elsewhere [10].

The levels can be clearly aligned by students with different volumes of data. Typically, system RAM goes up to the size of several GB, while in disks there is an scaling to near some TB, and then clusters can scale up. Years ago, the complexity of moving from (b) to (c) or (d) required specific programming, but this is becoming largely transparent in some cases thanks to so-called “Big Data” technologies as Apache Spark and others.

The second, related level is that of paralelism, that is tied to the different levels. Processor-level parallelism involves in-core acceleration, and it is specially relevant to matrix computation, since it is at the core of GPU acceleration. Other forms of paralelism involve in-memory (thread or processes) and then distributed computing in a cluster. Programming models as those of Dask[1] or Apache Spark[2] have made the last two levels transparent to a large extent, and in-core paralelism is also independent of the code, as it will be described later.

The transition to (d) can be grasped by some basic understanding of the idea of *task graphs* which are directed acyclic graphs of partial computation. These, as the one in the example in Figure 1 generated with Dask[3] for the fitting of an estimator, are the basis for distributing computation across processees or nodes. Here the mental models needs not be detailed (and as such students are

[1] https://dask.org/
[2] https://spark.apache.org/
[3] https://jcrist.github.io/dask-sklearn-part-2.html

just exposed to simple examples as distributed computations of averages), as they concern internal details of the breakdown of computation. However, a basic understanding of the idea that vectorized computations of arbitrary complexity can be decomposed for distribution in different processing units in that way is required to avoid naive unscientific beliefs about the underlying mechanisms.

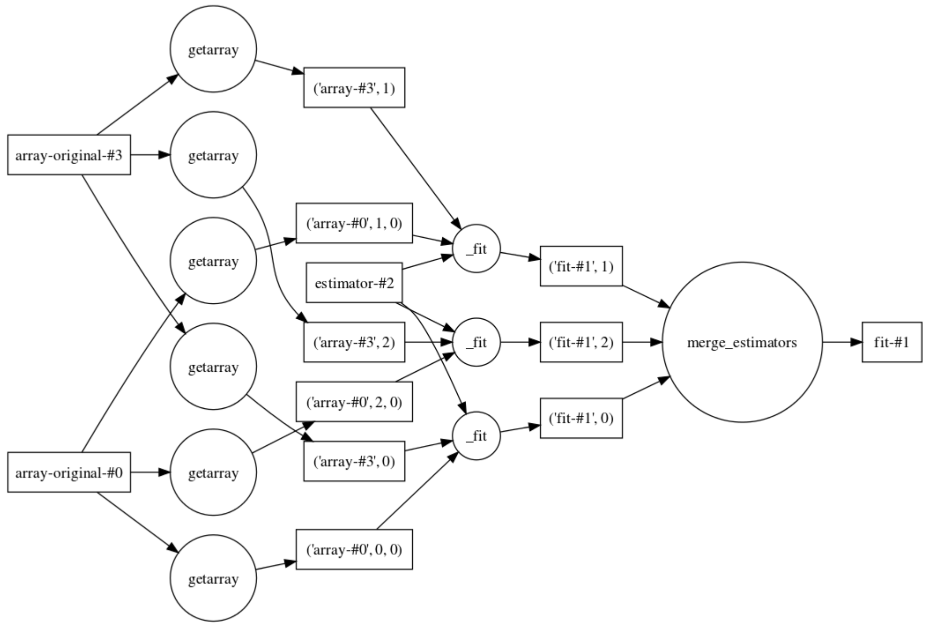

Fig. 1. Examples Dask task graph.

3.3 Streams of data

Dealing with streams of data requires a change in the way of thinking, as data becomes a continuous flow and the programs wait and react to the arrival of data. This is opposite "data applied to functions" to the idea of "functions applied to data" and thus requires separate treatment, typically introduced afterwards. While streaming can be understood as processing in batches of windows [4], the mental model has to focus on the asynchronous and infinite nature of data. That is properly reflected on the use of the dataflow paradigm, and depictions as "marble diagrams" as commonly used in discussions of reactive programming. Figure 2 shows a depiction of an event stream and the temporal transformation effect of a `filter` function[4].

[4] http://reactivex.io/documentation/operators/filter.html

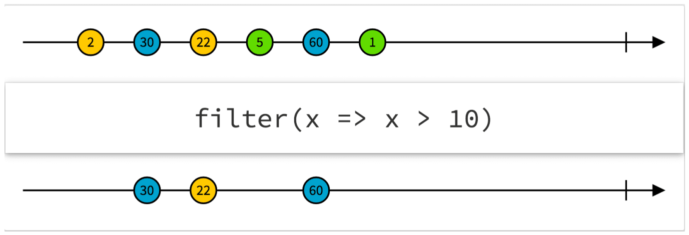

Fig. 2. Example reactive "marble diagram".

An important observation is that thinking on transformations on streams fits the functional paradigm in that operations are typically expressed as purely functional primitives. Further, this functional style can also be applied to non-streaming data thanks to current unifying frameworks [3] as implemented in Apache Beam[5]. For that reason, a model for streaming data can be considered an extension of a data processing functional model in which new concerns are added, e.g. that of windowing or temporal delays.

4 Fundamental paradigms

A programming paradigm is "an approach to programming a computer based on a mathematical theory or a coherent set of principles"[20]. Each paradigm supports a set of concepts that makes it the best for a certain kind of problem or task. The consideration of the mental models described before is actually orthogonal to the three key programming paradigms we have found as important and are discussed in what follows. This is because all of them abstract out memory hierarchy, are able to somewhat perform automatic paralelism across differen levels of that hierarchy, and in some cases allow for dealing with both streaming and non-streaming data in a declarative style. Further, all of them make extensive use of vectorization as a basic primitive. Then, the difference among them is the key building block used in the code, that is discussed in what follows.

4.1 Lower level abstractions: functional array/bag processing

The first concept is that of working with homogeneous arrays of data, with one or several dimensions. These can be thought of as arrays or bags (understood as generic sequences of elements), and data science languages, due to vectorization and broadcasting, can be taught in a functional style. In `numpy`, for example, *ufuncs* have a functional common interface.

```
x = np.arange(1, 6)
np.add.reduce(x)
```

[5] https://beam.apache.org/

The problem with a full functional paradigm is that pure functions are by definition stateless, and efficiency matters require exposing students to in-place modification semantics. An expression such as `a *= 2` corresponds to an in-place operation, where all values of the array are modified. By contrast, `a = a*2` means that a new array containing the values of `a*2` is created. The same occurs in `Series` in which the parameter `inplace` is for example used to change the memory management. Also, slices (which can be seen as a filtering operation) are by default views and not copies, again exposing memory management to programming constructs.

A more complete functional paradigm at this level is provided in `bag` libraries of Dask, a multiset library that is able to operate with data out of core and in clusters. The following is an example processing Json data. It is equivalent to Apache Spark RDD interface[6].

```
result = (b.filter(lambda record: record['age'] > 30)
          .map(lambda record: record['occupation'])
          .frequencies(sort=True)
          .topk(10, key=1))
```

As it can be seen, the functional paradigm is a viable alternative for most of array computations in data science frameworks. This is a key practice as leads to more concise code and fits better with other paradigms as dataflow that will be discussed later. The main problem is that current main languages, Python and R still require the student to be aware of buffering, in-memory operations and views, since the languages are imperative and not functional, and writing efficient code requires knowledge of those details.

4.2 Dataframe-centric manipulation

The second model introduces the mental model of a *dataframe*. This is essentially a collection of one-dimensional series that share a common index (which is represented typically at the "rows"), and are themselves indexed by name (represented in the "columns"). That is a pervasive notion of data collections, since for example, machine learning libraries assume a tabular representation of this form, with rows as instances and columns as variables or features. However, that notion of dataframe is actually used for holding different forms of data, and the schema that is implicitly required by many machine learning libraries is corresponding with the notion of "tidy data" [23].

Tidy data can be assimilated to normalized relational databases and it is a concept that helps in differentiating the typical structure of a machine learning problem from other types of data encoded in dataframes. As such, the teaching of the dataframe paradigm could be split in several "forms", that of tidy data and others that are untidy and thus may require transformation.

The dataframe paradigm for tidy data then reduces to bag processing when taking a simple column, or to something close to the notion of relational table

[6] https://spark.apache.org/docs/latest/rdd-programming-guide.html

data otherwise. Actually, for students knowing SQL it proves useful to introduce dataframe manipulation using an analogy to SQL first. For example, in pandas: (a) boolean indexing can be assimilated to `WHERE` clauses, `merge()` for joins, and `groupby()` to the corresponding `GROUP BY` SQL clause. Then, multi-indexes, pivot tables and other advance or more flexible behaviour can be introduced as additional features to what is available in SQL.

A dataframe-centric approach emphasizes thinking on state (collections of tables) instead of on functions, even though the manipulations can be done applying vectorized functions. A typical interaction in an interactive environment as a Jupyter Notebook is typically a sequence of operations that modify, create views or provide data to libraries taking fragments of an in-memory dataframe. Memory hierarchy and paralellism are becoming largely transparent, e.g. with the abstraction of a Dask `dataframe` that provides an interface that is analogous to that of a regular pandas `dataframe`, as shown in the example in which `compute()` triggers the start of the computation of the expression `t` that is represented as a task graph.

```
import dask.dataframe as dd
df = dd.read_csv('files/2018-*.*.csv', parse_dates=['timestamp'])
t = df.groupby(df.timestamp.dt.hour).value.mean()
t.compute()
```

However, the technicalities and details of those are still surfacing the programming tasks. An example is the need to specify *chunking* parameters, that determine how the underlying task graph divides the data for parallel processing. Another example is the use of different `schedulers`, i.e. execution frameworks, as can be appreciated in Dask's documentation, e.g.: "The multiprocessing scheduler is an excellent choice when workflows are relatively linear, and so does not involve significant inter-task data transfer as well as when inputs and outputs are both small, like filenames and counts.". This still requires a mental model of the memory hierarchy and some operating system concepts.

4.3 Pipelines of data processing

The third model has the notion of a pipeline as the central processing element, and data becomes the input-output of a computation expressed as a directed graph of stateless operations that perform their task as data becomes available. These *dataflow* languages are inherently parallel and can work well in large, decentralized systems.

This paradigm requires a change in thinking from a dataframe-centric approach, since programs are activated by incoming data and do not follow the idea of a typical von-Neumann architecture. Reactive programming is considered a kind of dataflow that emphasizes the fact that the computation is continuous and triggered by external events. The following is an example word count for a stream of data using Apache Beam Python SDK.

```
counts = (lines
| 'split' >> (beam.ParDo(WordExtractingDoFn())
.with_output_types(unicode))
| 'pair_with_one' >> beam.Map(lambda x: (x, 1))
| beam.WindowInto(window.FixedWindows(15, 0))
| 'group' >> beam.GroupByKey()
| 'count' >> beam.Map(count_ones))
```

As it can be seen, the different stages are composed with pipes, and follow well-known functional primitives. The new element that needs to be added is that of windowing, which introduces concerns for data flowing with some delay. This model can be used at the levels of array/bag processing as in the previous example, but could also be mixed with dataframe style manipulation as in the following Apache Spark example in Scala.

```
people.filter("age > 30")
.join(department, people("deptId") === department("id"))
.groupBy(department("name"), "gender")
.agg(avg(people("salary")), max(people("age")))
```

In both cases, the notion of pipeline is sufficient to introduce streaming elements into the previous paradigms seamlessly. These constructs are sufficient to convey the mental model of a stream of data that can be bursty and is virtually infinite, i.e. data that triggers computation.

5 Other relevant aspects

Here we describe additional contextual or related elements that are critical to provide an understanding of some decisions students face when selecting execution frameworks or libraries in particular situations.

5.1 Optimization

Another key decision is that of understanding how immutable structures affect optimization, and the implications of the functional paradigm in so. The two key ideas that require specific instruction are that of static typing and optimization. The idea that statically typed languages as C/C++ provide more information to code optimizers is fundamental and difficult to grasp by students with no CS background. However, it is required to understand why for example in Python, the Cython super-language is used, or even direct call to C or Fortran libraries. This is also key to understand approaches to provide hints for optimization. An interesting example is that of the library `numba` that translates Python functions to optimized machine code at runtime using the industry-standard LLVM compiler library. A typical decoration of a function with `numba` is as follows.

```
@vectorize(['float32(float32, float32, float32)',
'float64(float64, float64, float64)'], target='cuda')
def cu_discriminant(a, b, c):
   return math.sqrt(b ** 2 - 4 * a * c)
```

This provides a way of understanding the process of optimization that is related to computer architecture, and links with the idea of parallelism at the microprocessor level.

5.2 Understanding costs

A typical problem in deciding the computational framework is that of the cost of maintenance and use of the infrastructure. While using a single-computer configuration, the costs are considered transparent, but the decision to switch to a cluster requires an understanding of the total cost of ownership (TCO), which is a complex topic and would require careful consideration of diverse factors [22]. The pricing of cloud computing or *on-premises* equivalents have become more complicated due to the widespread adoption of specialized hardware for particular computing problems, as in the case of training deep learning models [19].

All these hardware and infrastructure needs impact the decision on using some libraries or others, and require that data science students to have some basic knowledge of computer and network architecture. There are reports of using such kind of infrastructures [12] but not about specific instructional designs. At least a basic understanding of complexity notation and "big-O" classes of complexity is required so that students can reason about decisions on choosing one or other algorithm or approach.

5.3 Sequencing and context

Sequencing is fundamental to instructional desing, and it can be argued that the three models may be introduced using different paths. We have experimented an "array *then* dataframe *then* pipeline" approach as a layered approach that presents constructs as compositions of the previously presented ones. A possible alternative route is that of using a streaming-first approach, as it can be argued that data-at-rest is a particular case of data-at-motion [4].

A related aspect is that of providing students with the adequate context. This depends on the discipline or degree, as it should be different if the students are in a bioinfirmatics degree than in a business analytics program. In our case, the context falls in the latter, and required adapting cases and examples so that the training is not disconnect from the rest of the program, as typically real-world or realistic analytic cases come after the training focused on programming, data acquision, cleaning and preparation. Our experience for our case shows that an approach to profit-driven analytics (see for example a case of churn in [14]) is key from the beginning. A case in which for example, a classifier needs to be selected by considering the cost of the errors provides the context to link business or domain context to technical work. This can also be discussed by situating the training in the framework of a process model as CRISP-DM.

6 Conclusions and outlook

Teaching programming for computational science using modern data science frameworks requires a careful consideration on mental models, programming paradigms and other aspects that affect the task of coding for data wrangling or analytics.

We have discussed the findings on teaching computational science programming to ten cohorts of students of data science-related programs, coming from diverse, non-computer science backgrounds. We followed an action research framework, that included new elements and variations in different subsequent cohorts and evaluated the effect of these new elements through examining outcomes of assignments and interviewing students. The mental models discussed come from direct experience in facing student difficulties when understanding code and when translating the idea of code running on their laptop to similar code running as a sequence of paralell tasks, possibly in a distributed setting on a cluster.

We argue that there are three key fundamental programming models that should be dealt with separately for a complete understanding and effectiveness in leveraging current data analytics technology and libraries. We have discussed their main implications and relations, and the approach to introduce them as layers that add complexity. Essentially, array/bag models can be taught as purely functional, before introducing the dataframe centric approach and then a pipeline model that fits the additional constructs required to deal with streaming data. It is possible to abstract out the primitives from the concrete language, library or execution framework, which allows for a significant degree of transfer of learning, that is made evident when students move, for example, from a local Python environment to a distributed setting using Apache Spark.

The results discussed as subject to inherent limitations including type of program (oriented to business analytics) and the characteristics of the participants. However, we believe the discussion can be used as a basis for contrasting other experiences and delineating research directions. Future work should deal with specific understanding problems and the transition and effectiveness of the mental models suggested here. Further, controlled experiments should delve into the details of particular programming constructs, idioms or other forms of expression.

References

1. Ahmad, A., Feng, C., Ge, S., & Yousif, A. (2018). A survey on mining stack overflow: question and answering (Q&A) community. Data Technologies and Applications, 52(2), 190-247.

2. Ambler, A. L., Burnett, M. M. & Zimmerman, B. A. (1992). Operational versus definitional: A perspective on programming paradigms. Computer, 25(9), 28-43.
3. Akidau, T., Bradshaw, R., Chambers, C., et al. (2015). The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. Proceedings of the VLDB Endowment, 8(12), 1792-1803.
4. Akidau, T., Chernyak, S., & Lax, R. (2018). Streaming Systems: The What, Where, When, and How of Large-Scale Data Processing. O'Reilly Media, Inc.
5. Baškarada, S., & Koronios, A. (2017). Unicorn data scientist: the rarest of breeds. Program, 51(1), 65-74.
6. Brunner, R. J. & Kim, E. J. (2016). Teaching data science. Procedia Computer Science, 80, 1947-1956.
7. Cañas, J. J., Bajo, M. T., & Gonzalvo, P. (1994). Mental models and computer programming. International Journal of Human-Computer Studies, 40(5), 795-811.
8. Chuprina, S., Alexandrov, V. & Alexandrov, N. (2016). Using ontology engineering methods to improve computer science and data science skills. Procedia Computer Science, 80, 1780-1790.
9. Davenport, T. H. & Patil, D. J. (2012). Data scientist. Harvard business review, 90(5), 70-76.
10. Eijkhout, V. (2016). Teaching MPI from mental models. In Proceedings of the Workshop on Education for High Performance Computing (pp. 14-18). IEEE Press.
11. Giabbanelli, P. J. & Mago, V. K. (2016). Teaching computational modeling in the data science era. Procedia Computer Science, 80, 1968-1977.
12. Ivica, C., Riley, J. T., & Shubert, C. (2009). StarHPC—Teaching parallel programming within elastic compute cloud. In Proceedings of the 31st International Conference on Information Technology Interfaces (pp. 353-356). IEEE.
13. Kurgan, L. A. & Musilek, P. (2006). A survey of Knowledge Discovery and Data Mining process models. The Knowledge Engineering Review, 21(1), 1-24.
14. Maldonado, S., Flores, Á., Verbraken, T., Baesens, B., & Weber, R. (2015). Profit-based feature selection using support vector machines–General framework and an application for customer retention. Applied Soft Computing, 35, 740-748.
15. Malik, S. I. (2018). Improvements in introductory programming course: action research insights and outcomes. Systemic Practice and Action Research, 1-20. Springer.
16. Manieri, A., Brewer, S. et al. (2015). Data Science Professional uncovered: How the EDISON Project will contribute to a widely accepted profile for Data Scientists. In IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom) (pp. 588-593). IEEE.
17. McKinney, W. (2010). Data structures for statistical computing in python. In Proceedings of the 9th Python in Science Conference (Vol. 445, pp. 51-56).
18. Saltz, J. S. & Grady, N. W. (2017). The ambiguity of data science team roles and the need for a data science workforce framework. In 2017 IEEE International Conference on Big Data (pp. 2355-2361). IEEE.
19. Sze, V., Chen, Y. H., Emer, J., Suleiman, A. & Zhang, Z. (2017). Hardware for machine learning: Challenges and opportunities. In 2017 IEEE Custom Integrated Circuits Conference (CICC) (pp. 1-8). IEEE.
20. Van Roy, P. (2009). Programming paradigms for dummies: What every programmer should know. New computational paradigms for computer music, 104, 616-621.
21. Van Der Walt, S., Colbert, S. C. & Varoquaux, G. (2011). The NumPy array: a structure for efficient numerical computation. Computing in Science & Engineering, 13(2), 22.

22. Walterbusch, M., Martens, B. & Teuteberg, F. (2013). Evaluating cloud computing services from a total cost of ownership perspective. Management Research Review, 36(6), 613-638.
23. Wickham, H. (2014). Tidy data. Journal of Statistical Software, 59(10), 1-23.