Reification and symbolization

Reinhard Oldenburg

Angaben zur Veröffentlichung / Publication details:

Oldenburg, Reinhard. 2011. "Reification and symbolization." In *Proceedings of the 11th Koli Calling International Conference on Computing Education Research - Koli Calling '11*, edited by Ari Korhonen and Robert McCartney, 49–53. New York, NY: ACM Press. https://doi.org/10.1145/2094131.2094141.

Reification and Symbolization

Reinhard Oldenburg
Goethe University Frankfurt
Robert-Mayer-Str. 10, 60325
Frankfurt/M., Germany
0049 69 798 23770
oldenbur@math.uni-frankfurt.de

ABSTRACT
The construction of mental objects by learners is a very complex process and it is desirable to understand it as deeply as possible, especially to understand domain specific subtleties. In this paper we will argue that the adaption of the reification theory that has been used successfully in mathematics education provides new and important insights into the learning of programming.

Categories and Subject Descriptors
K.3.2 [**Computer science education**]: Computer science education

General Terms
Theory

Keywords
computer science education, construction of mental objects, genetic teaching method, literals

1. INTRODUCTION
Meaningful statements are often concerned with certain objects. Language is a referential system that relates objects. We speak about apples, numbers, objects, classes and protocol stacks. The theory of radical constructivism (von Glasersfeld [7]), which is widely accepted in didactics, frees us from the question (which appears nevertheless to be an interesting question from a philosophical point of view) whether these things exists in reality (or, what 'reality' should refer to in this sentence to make it true or at least viable). Mental objects are part of an individual's subjective ontology. However, successful communication between two individuals requires at least a partial mapping of their private ontologies. Thus, when speaking about mental models we are directly led to ontological questions. Similarly, formal logic has found another way to get rid of these ontological questions by shifting the specification of a domain to possible interpretations of its formulae. The picture sketched so far is by large compatible with the framework of Ontological Relativity [10] which will thus be adopted. Thus we are freed from the necessity to commit ourselves to decide what really exists. The assumption that certain objects exists is of hypothetical nature and thus we can focus on the question how these hypothetical objects come into being. This question is the important one from an educational point of view and this article will focus on it.

Students who learn about computers and computer science will have to create new mental objects. These objects form a very broad spectrum ranging from such objects which are close to physical objects build from sensual experience (e.g. the concept of a pixel), objects which can be located in space although they are not accessible visibly (e.g. the magnetic sectors on a hard disk that make up a file) and finally objects that cannot be located in space such as method or binary tree) Teaching experience suggests that even the middle category of these objects present problems to learners. Even more abstract objects must be created by a demanding mental effort. This process is error-prone and we hope to shed some light on it.

The next section will introduce the central concept of this paper, i.e. reification. This concept deals with the creation of mental objects and is described in several theories which may, due to their large mutual overlap, be combined (at least for our purpose) into one theory. The use of reification theory has proved to be useful in mathematics education and we hope that computer science education is a similar fruitful ground for its application.

1.1 Reification
As didactics of computer science education is a relatively new science it can try to learn from more mature disciplines like mathematics education. Of course, the differences of their domains should not be ignored, and each idea transferred from one domain to the other has to be investigated individually to check if its validity survived the domain change.

Mathematics education has discussed for some time the question how abstract mathematical objects can be (re)constructed by the students. In this course a number theories proved to be helpful that may be combined under the heading of reification theories. There are some important differences between some of these theories, but nevertheless we simplify things by presenting them as one major theory.

The word ‚reification' is based on the Latin word ‚res' for thing. A very influential use of this concept was given by Anna Sfard ([12]). She starts from the thesis that the cultural and historical development on one hand and the individual development on the other hand show strong similarities. Hence, the study of the historical genesis of some relevance for didactics and Sfard looked especially on the development of concepts that took a longer period before they were cast into their final form. An example is given by the complex numbers. Their name hints already at the difficult genesis they have had. For quite a long time, mathematicians considered them to be suspicious – in

Koli Calling '11, November 17--20, 2011, Koli, Finland.

contrast to the real numbers they had been used to. One may start to learn how to work with the symbol *i* that shall represent the square root of -1 and may get some routine in doing the calculations with it, but really seeing *i* as a number requires a conceptual change of what a number is. (see [14] for conceptual change). Before the introduction of imaginary numbers, all numbers can be represented on the number line and hence can be ordered. This useful property is no longer valid in the complex number field and hence many operations with numbers lose their basis. Sfard looked at this and other examples and studied how the process of reification works, i.e. how new mathematical objects come into being.

A geometric concept, e.g. the concept of circle, may be learned more or less directly by ostension, i.e. by exposing examples and abstraction [12, p. 10], most concept formations however, start with processes. E.g. the natural numbers develop from the process of counting. Fractions have historically been introduced as means for measuring, i.e. a/b meant the number of times b is used to form a. This measuring process is likely not to work out in the most simple form that there is a natural number n such that $b=n\cdot a$, but more generally ancient Greeks allowed the understanding that there is a common measure e such that $a=n\cdot e$ and $b=m\cdot e$, thus this measurement process is condensed and forms a rational number [12, p. 11]. Consequently, the Greeks interpreted irrationality of $\sqrt{2}$ as the sign that measuring process cannot be carried out. But even processes that cannot be carried out can be reified to become an object. This development, however, took very much time in the history of mathematics. Learners as well need some time to make the double passage, first transforming the measurement process into the objects of rational numbers and then transforming even more measurements into irrational numbers.

Sfard looked closely at this development from processes to objects and found three steps which we illustrate with examples from computer education:

- Internalization: A process on objects that already exists is carried out and internalized so that it can be re-played mentally.
    - Example: Within an image manipulation program students repeatedly select subsets of pixels by using various tools.
- Condensation: The process is condensed so that it forms a new autonomous unit. This can be supported by a new form of notation.
    - Example: The students realize that, independent of the method of selection, the result is just a subset of all pixels. This set may be denoted by S – the current selection.
- Reification: The description of the process turns into one mental object that can be manipulated and handled in thought processes
    - A selection is a mental object that can be operated on, e.g. extended or reversed.

In this setup Sfard says that processes and objects are dual to each other. This is a step from the Piagetian Framework which clearly identified the role of internalization of operations in developing schemata but gave little attention to the dual picture. Even more emphasis on the connection between these kinds of mental entities is given by Gray&Tall [8]. They have coined the term *procept* to denote the combination of a process and an object. This makes clear that the newly born objects still contain the process and the precise meaning of the objects is defined by this process. At times it may be necessary to go back and invoke the process again, e.g. it after having being reified fractions are objects but they still contain the process of division and maybe this process will be evoked in certain situations.

The point of view that processes lead to objects is by no means new to computer science.

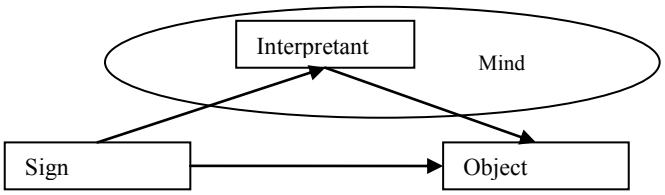
- Procedural abstraction turns processes into procedures. Example: When working with the Logo programming language, a child may discover that a certain process, such as drawing a regular polygon is useful over and over again and thus may write down a generic copy of this process in the body of a function definition.
- Abelson/Sussman [1] show how the data type of a list may be implemented on the basis of lambda expressions. This illustrates that the distinction between (static) data and (dynamic) procedures is blurred by little more than the most basic understanding of a programming language.
- Consider the example of the meta-circular implementation of a Lisp interpreter as is done in most Lisp books. This interpreter may be seen as an abstract machine that can do arithmetic and list processing. Variables in such an interpreter are symbols that are looked up in environments, i.e. variables initiate the lookup-process that determines their value. Usually, the lookup process will signal an error, if there is no value stored in the environment for a given symbol. Interestingly, the simple modification that does not carry out this process where it is not possible, i.e. simple taking the symbol itself as result in this case and allowing arithmetic operations as well to reproduce themselves if not all operands are numbers, turns the interpreter into the basis of a computer algebra system. This shows that the passage from arithmetic to algebra may be taken – on a technical level – by not-carrying out certain processes. The computer algebra system muSIMP/muMATH (The Softwarehouse / Microsoft) used essentially this implementation strategy.

At this stage we can draw the first conclusions for the realm of education. When students are expected to construct new mental objects, they need to have enough time to carry out the relevant processes many times so that internalization and condensation can take place and prepare for the final reification step. It seems plausible that a better understanding of reification and of mental tools that support it may help in designing better learning environments and teaching strategies. To us it seems that symbolization (compare the role of notation in the step of condensation) is crucial.

1.2 Symbolization

The importance of symbols for thinking is obvious. Proper symbols help to reduce complexity and facilitate a playful interaction. Hence there is a long standing tradition of didactics in the use of symbols and especially in semiotics as the discipline that is crucial for the understanding of symbols. The following short exposition follows Filloy et al. [5]. A simplified understanding of symbolization may suggest that one has a sign S that refers to an object O, symbolically $S\rightarrow O$. This simple form is useful in many places but not in all. The extension of this to the semiotic triangle introduced by Peirce gives room for the

individual person that makes the connection between sign and object:



A sign (Symbol) refers to an object and is directed to a person. In the mind of this person a new mental sign is established that points to the objects as well, it is called the interpretant. This model gets its power but also its complexity from the fact that the relation can be iterated. I.e. the interpretant can turn into a sign as well, e.g. when written down. The French philosopher Lacan has considered the other direction as well: A sign turns into the object for another sign. Luckily, for the rather simple applications we have in mind this is not strictly necessary but it should be kept in mind that one gets involved in a non-trivial net of relations.

Signs, objects and interpretants in this semiotic triangle can be of different kinds. Peirce distinguishes between different kinds of signs, e.g. between signs that are similar to the referred objects and signs that are purely conventional. Signs may be compound signs that are composed of other, more elementary signs. Compound signs are expressions, UML diagrams and much more. This fact will be important for us. Similarly, the objects can be specific ones (a table, the number 5, memory cell $ffaa7b,…) or generic ones (strings, classes,…). The nature of interpretants is consequently very diverse as well. It seems to be reasonable to view certain compound signs as mental models [9] but this point of view seems to be present in the literature and we won't rely on it.

Sfard [13] takes semiotics as a basis for understanding the creation of mental objects. To understand this, the following facts must be understood: Signs can be names, symbols or graphical representations. Compound signs are allowed to make up structures containing other signs, possibly compound as well. Special compound signs are those that refer to on an object that structured in way similar to the sign. They are called ‚structural signifier'. Such signifiers are the plus sign in n+1 or operators in programming languages.

Sfard claims that the development of a new structural signifier means the development of a new mathematical object. The character string ‚3+4' can be read as prompt to do a calculation. Reification means however, that this process is not carried out but condensed to a new form. This new form, which may be written with help of the structural signifier '+', is an arithmetical expression – a kind of object that may not have existed in the individuals mind before. Maybe even more enlightening is the structural signifier that denotes surds and especially the root of -1. The structural signifier √ defines new numbers out of known ones.

Structural signifiers allow one to write down composed objects and operate on them or their parts on paper. This can be used to execute processes on these objects and these can in turn be the starting point of a new reification cycle. This point of view coincides with that of W. Dörfler [4]. From a semiotic point of view he points out that making meaning from diagrams creates mathematics. In this sense, the signs on paper don't refer to mathematical objects that exist independently but these inscriptions *are* the mathematical objects.

The above subsections should have made clear that within the mathematics education research community it is consensus that inscriptions and especially structural signifiers create mathematical objects at least mentally if not in every sense of existence. The relevant point is that inscriptions allow one to carry out processes more complex than processes that can be completely simulated in mind.

2. Reification and Symbolization in the learning of programming

Based on the understanding of learning processed outlined above one can easily justify well-known principle of good software design, i.e. one may explain why certain design properties eases the user's learning experience. Graphical user interfaces provide a multitude of structured symbols (e.g. the directory tree) and these generate their interpretants. The same holds true for the blocks in the Scratch programming language. However, the example of mathematics shows that textual representations can be the starting point of successful reifications as well. This has a lot of in common with the learning of programming and we will explore this connection in more detail in the following sections.

2.1 From processes to objects

There are so many examples for the creation of objects from processes that we restrict ourselves to some examples.

Example Design Patterns: During programming students will notice that several situations appear over and over again and that similar situations trigger similar processes. E.g. if the collection of data (a list, set, tree,…) is given, this data structure is to be traversed, i.e. an index variable is defined and used to walk thru the dataset. Repeating this design-time-process several times may lead to create the iterator design pattern as its object. Other design patterns from [6] can be understood in a similar fashion.

When drawing figures with turtle graphics one first has drawing processes that are repeated all over and finally a mental object „square" and a procure realizing it are constructed.

Queries into a database are at first a process that takes place, but step by step they get reified, one can talk about former queries that are repeated or queries that are stored in a queue to be processed.

2.2 The role of literals

A literal is a way to write down in source code a description of an object which is constructed from this description. (cf. [11]), i.e. it is an alternative to using the constructor of a class.

Interactive programming languages typically support literals for a rich range of object classes. In Python one has literals for numbers (integers, long integers, floats, fractions, complex numbers), strings (e.g. 'Text'), lists [1,2,3],[["a",1],["b",3]]), hash tables (dictionaries, {„a":1, „b":2}), tupel and (with some restrictions) anonymous functions (e.g. lambda x: x+1).

Without the above explained background theory it may appear that literals are simply a way to make life easier for the programmer, but in the light of this theory it should be clear that literals have an important role in forming mental models of the objects they describe.

Bennedsen&Schulte [2] describe a competence test on understanding of object oriented program constructs. One item in this test asks students to predict what the output of a loop will be that iterates over the list constructed in the following lines::

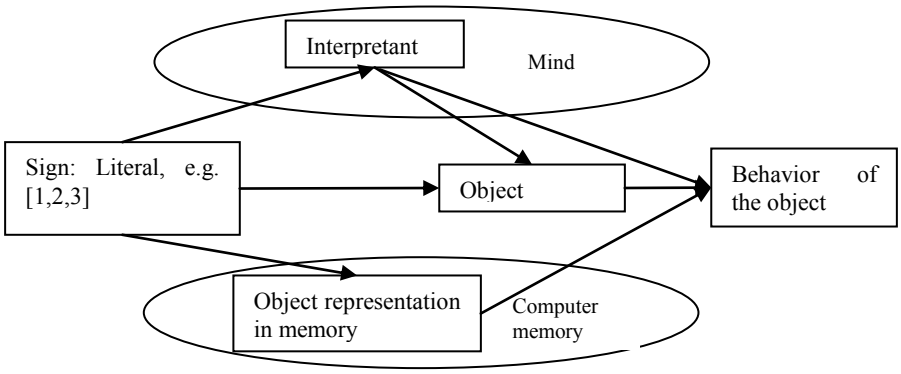
```
List<A> l= new ArrayList<A>();
l.add(new A(76));
l.add(new A(-10));
l.add(new A(6));
l.add(new A(43));
```

The isomorphic structure as a literal is just `l= [76, -10, 6, 43]`, which is not only shorter but can be interpreted more easily. as a compound sign that consists of structural identifiers (brackets and commas) and inner objects (numbers). The structure of this compound sign can be takes as the template of the mental model of a list. It clearly suggests what can be done with objects of this kind (e.g. determine the number of objects contained in it or determine whether a certain object is contained in it.) These operations can be carried out even on paper using the inscriptions. With a bit of flexibility this extends to destructive list operations such as deleting an element from the list. Thus all processes typical for lists can be carried out on this paper model and hence the paper model can be reified easily to the mental object of a list.

It should be emphasized that this works out so well because of the use of structural signifiers that have already got a meaning from different contexts: The comma as a separator signals that the individuals objects are just grouped not joined tightly. Without such helping knowledge learning is harder but not impossible as Sfard has shown in Sfard [13]. She conducted an experiment where a test person was confronted with strange meaningless symbols that where manipulated according to certain –at first sight – strange rules. Nevertheless the test person readily came up with a meaningful (hypothetical) interpretation. This shows that human minds a preconfigured to make sense out of symbols.

2.3 The extended semiotic triangle

When working with a computer the above described semiotic triangle is no longer an adequate description. The computer is another medium with its own inscriptions and with an own internal state. The following figure is a suggestion how this may be taken into account. The symmetry of this figure shall indicate a rough analogy but by no means shall it express equality between these two areas.



The literal creates an interpretant in the user's mind an as well an internal representation in the computer's memory. The latter representation is not passive but may be acted upon by processes. This shows a non-trivial behavior of the object that can be observed by the user. The user can draw conclusion about the object both from its mental interpretant as well as from the behavior displayed in the work with the computer. These two lines of conclusions are bound to yield the same result. Otherwise a misconception of the user (or a bug of the computer system) has been detected. Thus one has a very quick and effective way to synchronize the mental model and the computer model.

One may ask, if in the above diagram the object in the center is necessary at all. Wouldn't it be possible to say that the user's interpretant refers to the object in the computer (resp. to the representation there)? However, I feel that this would not be really useful: When a list is created from the literal [1,2,3] in Python and in Ruby then it is sensible to say that the internal representations created by reading in these literals (as different as they might be) represent the same object. And to do this, one should assume the existence of an abstract object 'list'. This point of view highlights the hypothetical and theory-loaden status of objects. Objects do not merely exists but emerge from a complex process of building an ontology that is rich enough to support all things we want to say over the world, without being too rich.

2.4 Diagrammatic thinking

Literals for lists are a notation form that is rich enough to support diagrammatic thinking in the sense of Dörfler. One may say that literals offer a way to create inscriptions that is rich enough to model universal structured data (e.g. stack, queues, association lists, trees, …).

As an example we suppose the task set for the students is to describe the books in the school's library. For this purpose one has to invent data structures and using literals this process can be done simple by tentatively writing down – and eventually adjusting – what might be a prototype of the data. It may look like this:

```
Books= [ ["Author", "Title","ISBN","Publisher",
…],
         ["Frisch", „Stiller",988353, "Insel",…],
         ["Eco", „The name of…", 778353, "BI",…],
                …
          ]
```

The operations that can be performed on data like this can be mentally tried out on tis prototype and subsequently can be programmed. We expect that students who are used to writing down structures in lists and who have mastered basic algorithms can come with basically all the operations that make up relational algebra in such a context.

Furthermore we suppose that the use of literals mediates between formal understanding of concepts and their application. In interviews with students we have observed the following counter-example of this hypothesis: The students had just completed a unit on simply-chained lists which had been implemented in an OO style as compose of objects from a class „Pair" that holds a data element and the rest of the list. The interview showed that all students had understood this structure, could explain it and could work with it. However, none of the students was able to apply this knowledge to the following modeling task: "During a jumping contest each students of a class is allowed to jump three times. The values are recorded and stored in a computer together with the name of the student. Can you design data structures that can handle this kind of data?" Obviously, students did not fail because they had not learned their unit well enough, but because the formal description of lists is not useful in application contexts, i.e. it does not provide an adequate mental model that can be used to

judge here this concept is sensible. We strongly expect that students who use literals for lists can more easily adopt this tool to model such situations. A teaching experiment to test this hypothesis is underway.

2.5 Pseudo literals

The positive effect of literals is not restricted to lists. However, we will not go on to discuss dictionaries which are rather similar, instead we'll take a look at anonymous functions.

Python allows one to write down anonymous functions using the lambda notation as `lambda x: x+1`. This describes the function that adds one to its argument, i.e. `(lambda x: x+1)(7)` yields 8, just like `f=lambda x: x+1; f(7)`. This is an example of a literal that encapsulates a process in the sense of reification theory (the addition) into a new object (the function). The advantage of this function is that it is a first-class object in the programming language, i.e. it can be stored in data structures and passed to and from other functions. It is an important goal to achieve the same flexibility with mental objects as well!

In general one has to be cautions because here we don't have literals in the same sense that we used before. The written form does not encode all information captured in a lambda function: The literal `lambda x: x+a` describes different functions depending on the context in which it has been written down. This context defines the reference of `a` that is used when evaluating the function body. (lexical scoping). This is the reason we prefer to call this a kind of literals pseudo literals.

2.6 Misconceptions

The hypothesis that the written expression determines the structure of the mental object explains some misconceptions of students.

Students often have difficulties to grasp semantic differences if they are not represented visually. As an example we look at programming JavaScript in web pages. : When a number is entered into an input text field, students typically expect that `name_of_the_form.field_name.value` is a number (o refers to a number) that can be used in calculations. However, the value of this value-attribute is string in this case which must first be converted into a number. The difference between string and number is not made clear enough by the notation in this example and hence students don't get the right picture.

One may observe that objects that cannot be described by literals are especially hard to learn. Without literals the mental construction has to rely directly on processes carried out with these kinds of objects. Examples are cyclic lists or sets of objects with mutual associations.

3. Conclusion

We presented some ideas from semiotics and mathematics education and applied them to computer science education. We hope that the resulting improved understanding of the learning process will open up the way for a teaching style that takes the learning trajectory into the focus. An on-going teaching project on 'genetic computer science education' tries out these ideas in practice. First results are encouraging and we hope that teaching when based on these ideas will give students more confidence that they can master the complexity of computers and the science behind them.

Among the very concrete suggestions we make based on the outlined theory are the use of languages like Python or Ruby that support a large class of literals and allow interactivity. Moreover, we see strong support for the hypothesis that symbolization using literals can be understood as a form of diagrammatic thinking that eases modeling tasks. The empirical investigation of this is underway.

4. REFERENCES

[1] Abelson, H.; Sussman, G. J.: Structure and Interpretation of Computer Programs. MIT Press 1996.

[2] Bennedsen, J.; Schulte, C.: A Competence Model for Object-Interaction in Introductory Programming In: 18th Workshop of the Psychology of Programming Interest Group, University of Sussex, September 2006, PPIG 2006

[3] Cobb, V. M.; Cobb. P.; McClain, K. (Hrsg.): Symbolizing and Communicating in Mathematics Classrooms: Perspectives on Discourse, Tools, and Instructional Design. Lawrence Erlbaum, 2000.

[4] Dörfler, W.: Diagrammatic Thinking. In: M. Hoffmann et al.: Activity and sign – Grounding mathematics education. Springer 2005.

[5] Filloy, E.; Puig, L.; Rojano, T.: Educational Algebra. Springer 2008.

[6] Gamma, E. et al.: Design Patterns. Elements of Reusable Object-Oriented Software. Addison Wesley, 1995

[7] Glasersfeld, E. von (1995) Radical constructivism: A way of knowing and learning. Falmer Press: London.

[8] Gray, E.; Tall, D.: Duality, Ambiguity and Flexibility: A Proceptual View of Simple Arithmetic, The Journal for Research in Mathematics Education, 26 (2), 1994, 115-141.

[9] Johnson-Laird, P.: Mental models. Towards a cognitive science of language, inference, and consciousness. 6th print. Cambridge: Harvard Univ. Press, 1995.

[10] Quine, W. v. O.: Ontological Relativity. New York 1969.

[11] Schneider, U.; Werner, D.: Taschenbuch der Informatik. Hanser 2007.

[12] Sfard, A.: On the dual nature of mathematical conceptions. Educational Studies in Mathematics 22, 1991, 1-26.

[13] Sfard, A.: Symbolizing Mathematical Reality into being – or how mathematical discourse and mathematical objects create each other. In: [3]

[14] Vosniadou, S. (Hrsg.): International Handbook of Research on Conceptual Change. Lawrence Erlbaum, 2008.