An Exploration of Novice Programmers' Comprehension of Conditionals in Imperative and Functional Programming

Claudio Mirolo
University of Udine
Udine, Italy
claudio.mirolo@uniud.it

Cruz Izu
The University of Adelaide
Adelaide, Australia
cruz.izu@adelaide.edu.au

ABSTRACT

Students of introductory programming courses are expected to develop higher-order thinking skills to inspect, understand and modify code. However, although novices can correctly write small programs, they appear to lack a more abstract, comprehensive grasp of basic constructs, such as conceiving the overall effect of alternative conditional flows. This work takes a little-explored perspective on the comprehension of *tiny* programs by asking students to reason about reversing conditionals in either an imperative or a functional context. More specifically, besides deciding if the given constructs can be reversed, students had to justify their choice by writing a *reversing* program or by providing suitable *counterexamples.*

The students' answers to four reversibility questions have been analysed through the lens of the SOLO taxonomy. 45% of students correctly identified the reversibility for the four code items; furthermore, more than 50% of each cohort were able to provide correct justifications for at least three of their four answers. Most incorrect answers were due to failures to consider border cases or to edit the conditional expressions appropriately to reverse the construct. Differences in comprehension between functional and imperative languages are explored indicating the explicit *else* paths of the functional examples facilitate comprehension compared with the implicit else (no update) of its imperative counterpart.

KEYWORDS

program comprehension; reversibility; imperative vs. functional programming; CS1; SOLO taxonomy

ACM Reference Format:
Claudio Mirolo and Cruz Izu. 2019. An Exploration of Novice Programmers' Comprehension of Conditionals in Imperative and Functional Programming . In *ITiCSE '19: The 24th ACM Annual Conference on Innovation and Technology in Computing Science Education, July 15–17, 2019, Aberdeen , UK*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3304221.3319746

1 INTRODUCTION

Besides introducing language constructs and examples of related code implementations, most CS1 courses will focus their learning activities to cover other practical abilities such as tracing and testing programs, with little concern over how to approach computations in a comprehensive way.

Recent work [7] has investigated program comprehension at introductory levels using think-aloud interviews. This work used the abstract concept of reversibility, to force students to reason comprehensively about simple statements. If we adopt the "reductionist" perspective addressed in [13] for the language notation and operational semantics, mastery of meaningful programming endeavours builds on the ability to envisage a range of potential computation flows and their relationships with each other for each basic constituent part. The *reversibility tasks* considered in our study were conceived precisely to get insight into such ability, specifically for the key conditional constructs. Moreover, a peculiar feature of this type of tasks is that, to a large extent, they allow to disentangle the understanding of program behaviour from any specific application domain knowledge.

As an apparently trivial conditional statement got the lowest performance in [7], we replicated part of that work by testing students' ability to deal with reversibility of two *if* statements [1], confirming that half of the subjects did not recognise that a simple conditional like that shown in Figure 1(i) cannot be reversed. The present study elaborates further on the results of this preliminary work by analysing both "if" and "if-else" conditionals in an *imperative* context, as well as their corresponding constructs in a *functional* context. In imperative programs instructions may change the state of variables, while in functional programs there is no state but computations return results that may be reused.

In order to assess higher order skills, students should be new to the task, thus they were not exposed to the concept of reversibility in class. After a short description of reversibility in the exam paper, students were asked to decide if the given conditional statements or functions can be reversed or not; additionally, they were asked to justify their decision: if their answer was *Yes*, by writing the code that would undo the state or by defining the inverse function; if it was *No*, by providing suitable *counter*examples.

In particular, we addressed two research questions:

Q1. To what extent can CS1 students envisage the *overall effect* of a conditional construct?

Q2. Does the programming paradigm, *imperative* or *functional*, have an impact on the comprehension of conditionals?

The main contribution of this study is twofold. Firstly, it uses a new perspective to explore program comprehension, both in imperative as well as functional contexts, by engaging students to reflect comprehensively about potential computation flows. Secondly, it provides a first insight into novice programmers' ability to analyse basic constructs in such a way.

The rest of the paper is organised as follows: after outlining some background in Section 2, Section 3 describes how the empirical data was collected and analysed; Section 4 summarises the quantitative and qualitative results, which are further discussed in Section 5.

Pre-print of the paper (accepted manuscript) for the institutional repository and not for redistribution. See terms of the ACM Copyright Transfer Agreement.
Submitted to ITiCSE '19, July 15–17, 2019, Aberdeen, UK – Published article: https://doi.org/10.1145/3304221.3319746

Analyse the following code fragments and determine if they are reversible:

- if your answer is *Yes, reversible* (i.e. the command is reversible), write a piece of code to restore the original state of the variables.
- If your answer is *No* (i.e. it is not always possible to undo the effect), provide examples for which we cannot recover the original state.

```
(i)    // int x                    (ii)    // int x
       if ( x > 10 ) {                     if ( x > 10 ) {
         x = x - 1;                          x = x + 1;
       }                                   }

(iii)  // int x                    (iv)    // int x
       if ( x < 0 ) {                      if ( x % 2 == 0 ) {
         x = x + 1;                          x = 2 * x;
       } else {                            }
         x = x - 1;
       }
```

Figure 1: Task on reversibility: imperative test (Java).

Analyse the following procedures and determine if they are reversible:

- if your answer is *Yes, reversible*, define the procedure for the *inverse* function.
- If your answer is *No*, provide examples showing that there cannot be an inverse function.

```
(i)   ;  int -> int        (ii)   ;  int -> int        (iii)   ;  int -> int
     (define (p n)               (define (p n)                 (define (p n)
       (if (> n 10)                (if (> n 10)                  (if (< n 0)
           (- n 1)                     (+ n 2)                       (+ n 1)
           n ))                        (+ n 1) ))                    (- n 1) ))

(iv)   ;  char -> char
       (define (pd c)
         (cond ((char<? c #\0) c)
               ((char<? c #\9) (integer->char (+ (char->integer c) 1)))
               ((char=? c #\9) #\0)
               (else           c)   )))
```

Figure 2: Task on reversibility: functional test (Scheme).

2 BACKGROUND

In Piaget's theory of cognitive development, reversibility is a key step toward more advanced thinking [17]. It is connected to progress from using and manipulating symbols to looking at them at a higher abstraction level and making better use of logical thinking. Moreover, according to a neo-Piagetian perspective, Piaget's learning stages are relevant independently of age when approaching new knowledge domains [21]. Thus, if this framework can also account, at least partially, for students' cognitive processes, reversibility should be an appropriate tool to analyse the first steps in the development of programming abilities.

Tasks asking to reverse a piece of code were first proposed by Lister [10], as a device to test an "archetypal manifestation of concrete thinking" in novice programmers, and later by Teague and Lister [22] to assess students' ability to reverse short programs. More recently, as mentioned above, this idea has been developed in [7] and [1] by asking the students to reason about short programs in terms of reversibility.

More in general, reversibility can be seen as a tool to assess program comprehension. This broader topic has been addressed from a variety of perspectives, focusing on the role of different types of abstractions, including data and control flow [16]; on the divide between code tracing, reading and "chunking" abilities [6, 9, 11, 12]; on students' ways of classifying code fragments based on perceived similarities and differences [23]; on mental models of program behaviour [3, 18]; on the correlations between performance and types of annotations in the exam papers [15]; on the understanding of loops and nested loops [4]; on the issues connected with basic concepts of language notation and operational semantics [13].

That even simple conditional constructs may present a challenge to some students emerged, in particular, from the analysis of a huge dataset by Cherenkova et al., who reported that "a significant number [of students …] exhibit the common errors of failing to check the border condition or reversing the conditional" [5].

3 METHODOLOGY

In this section we describe the reversibility tasks and the rationale for using them in our investigation. Then, we outline the data collection process. Finally, we present the criteria underlying our analysis and our application of the SOLO taxonomy.

3.1 Tasks

The tasks examined in this paper, unconventional from the students' standpoint, are shown in Figure 1 (imperative programming) and in Figure 2 (functional programming).

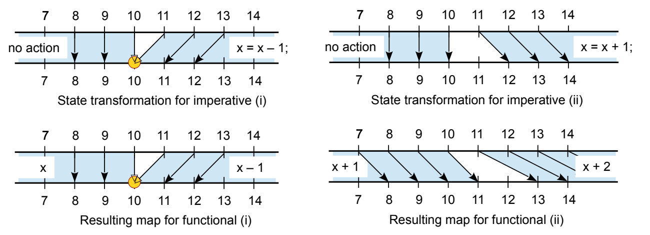
For the sake of our investigation, we selected a mixed method that combines the three item formats useful in measuring higher order thinking skills: *selection*, *explanation*, and *creation* [8]. For each of the four items, students had first to identify whether a program is reversible or not (selection); then they were required either to provide a counterexample (explanation), or to write a reversing program (creation).

In the exam paper the questions were preceded by a basic introduction on reversibility, not shown here due to space limitations.[1] Both tests were presented at the start of the exam paper when students were fresh and active. Moreover, for organisational reasons, each of the function definitions reported in Figure 2 appeared also in a second version of similar structure and, conceivably, of the same difficulty — and the order of the corresponding items was different in the two versions.

The operations in the branches of the conditionals are simple and individually straightforward to reverse. The key insight to decide about the reversibility of a given item is that there are different computation flows — one of which may be implicit — whose outcomes may "overlap," as shown figure 3 for item (i). In order to identify similar overlaps students have to think of and carefully analyse border computations. Then, if the program is reversible, to provide a correct solution it is necessary to deal with the relationships between condition and operations in the alternate branches.

An issue identified in the imperative task was that the (correct) code to reverse (ii) could fall in two distinct categories, depending on whether the student was aware of the fact the original condition could be left unadjusted or not. From a SOLO perspective [2], this makes it difficult to discriminate between relational, multistructural and unistructural answers. Thus, in the functional task we slightly modified the operations in (ii) in such a way that the condition should not be left unchanged when defining the inverse function (see Figure 3 for a visual illustration of both cases).

[1]Complete versions will be made available at a public repository.



State transformation for imperative (i)
no action
x = x – 1;
State transformation for imperative (ii)
no action
x = x + 1;
Resulting map for functional (i)
x
x – 1
Resulting map for functional (ii)
x + 1
x + 2

Figure 3: Depiction of the (state vs. functional) transformations for items (i) and (ii) in the two tasks.

3.2 Data collection

We collected the exam answers from two first year undergraduate cohorts at the University of *Town X*. Each exam paper was anonymised and digitized prior to analysis.

The first cohort took the imperative test in June 2017 as part of their CS1 exam. The test included six questions, of which we have analysed the first 4, and was worth 25% of the paper's marks. These 73 students had attended a full year of programming course (24 weeks: 80 lecturing hours + 60 practical hours) adopting a functional-first approach with Scheme followed by Java (imperative programming and basic notions of object-orientation).

The second cohort took the functional test in January 2018; the test also included six questions, and was worth 40% of that assessment. Note this cohort had only completed the first half year of functional programming with Scheme. The class was composed not only by students enrolled in the CS degree, as in the case of the first cohort, but also by student starting a new degree on emerging computing technologies. In addition, not all students progressed towards the final exam by June 2018. Thus, to provide a fair comparison between cohorts, the second dataset referred to in the next sections comprises precisely the 81 CS students who took also part in the June exam.[2]

3.3 Analysis

To begin with, we carried out a straightforward analysis to measure the percentages of students who had chosen correct options. Then, we conducted a qualitative analysis of the justifications — counterexamples or code — provided for each individual item. For the latter part, we referred to the framework of the SOLO taxonomy [2], a widely used instrument to classify the answers to code reading and writing questions, e.g. [12, 20, 24]. Although simple, the proposed tasks require abilities at the *relational* level that cannot be taken for granted in an introductory course.

Each individual answer has been mapped into four SOLO categories as per criteria listed in Table 1. To consistently apply these broad criteria, two researchers independently performed a deductive content analysis [14] to rate all students' answers to the imperative test. The discussions following this preliminary rating resulted into two subsequent revisions, and the overall inter-rater agreement of the second refinement was about 88%. Eventually, further discussion resolved minor rating differences and led to the final classification that is presented in section 4.2.

[2]The full functional cohort was composed of 170 students. Tables reporting the main figures for the whole cohort will be made available at a public repository.

Table 1: SOLO Classification guidelines.

SOLO Level	Answer features (reasoning or code)
Prestructural (1)	Poor answer showing either lack of understanding of the task or inadequate programming skills.
Unistructural (2)	Simplistic attempt to reverse part(s) of the code, while disregarding unequivocal interactions with other parts; attempts to reverse the code by some sort of cursory "syntactic manipulation"; flawed attempt to justify that the code cannot be reversed.
Multistructural (3)	Answer indicating that the goal of the task is clearly understood and pursued with a reasonable approach, but somehow incomplete, e.g. the reasoning may be ambiguous or the code may be affected by minor flaws.
Relational (4)	Correct and accurate answer, providing either some appropriate reversing program or a clear counterexample showing that the program cannot be reversed.

Table 2: Correct options and explanations for each item.

		Imperative			**Functional**		
item	reversible?	correct option	relational answer	SOLO mean	correct option	relational answer	SOLO mean
(i)	No	49.3%	37.0%	3.11	74.1%	64.2%	3.28
(ii)	Yes	97.3%	89.0%	3.73	82.7%	46.9%	2.85
(iii)	No	80.8%	58.9%	3.15	72.8%	65.4%	3.31
(iv)	Yes	94.5%	82.2%	3.66	82.7%	43.2%	2.69
both (i, iii)		47.9%	34.2%	—	63.0%	55.6%	—
both (ii, iv)		91.8%	79.5%	—	69.1%	27.2%	—

Based on this training, one researcher rated all functional test papers and produced very detailed guidelines, including most variants of justifications, that have then been checked and approved by a second researcher.[3]

4 RESULTS

4.1 Selected options

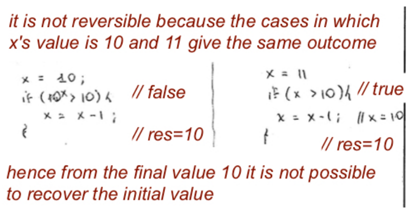
Table 2 shows the rate of correct options for the items presented in Figure 1 and 2. Although it may be expected that the imperative cohort would perform slightly better, as they had more instruction and practice, this turned out to be the case for (ii-iv) but not for (i). Relative to item (i), indeed, the functional group outperformed the imperative one, the ratio being about 3:2.

If we look at reversible (ii and iv) versus not reversible code (i and iii) we can see it is harder to identify non reversible items. There are two possible reasons: (1) answering *Yes* may be the default option for several students, and (2) it is difficult to spot an occasional flow overlap if not thinking on border values.

4.2 SOLO analysis

We next move to the SOLO analysis. Table 2 reports the percentage of relational answers and the SOLO mean resulting from the

[3]Also this document, with the detailed guidelines as well as the classification of several samples, will be made available at a public repository.



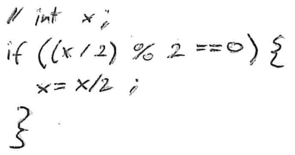
it is not reversible because the cases in which x's value is 10 and 11 give the same outcome

```
x = 10;
if (x > 10) {      // false
   x = x-1;
}                  // res=10
```

```
x = 11
if (x > 10){   // true
   x = x-1;  //x=10
}              // res=10
```

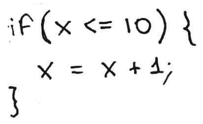
hence from the final value 10 it is not possible to recover the initial value

(a) Relational



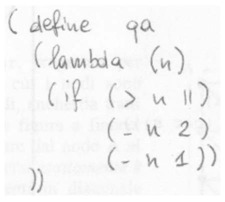
```
// int x;
if ((x/2) % 2 ==0) {
   x= x/2 ;
}
```

(b) Multistructural



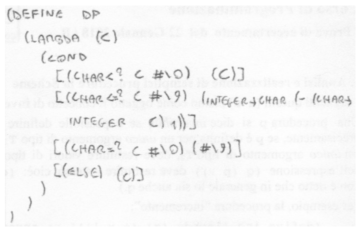
```
if (x <= 10) {
   x = x + 1;
}
```

(c) Unistructural



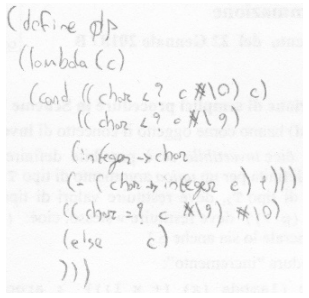
```
( define  qa
  (lambda  (n)
    (if  (> n 11)
         (- n 2)
         (- n 1))
))
```

(d) Relational



```
(DEFINE  DP
  (LAMBDA  (C)
    (COND
      [(CHAR<? C #\0)   (C)]
      [(CHAR<=? C #\9)  (INTEGER→CHAR (- (CHAR→
        INTEGER  C) 1))]
      [(CHAR=? C #\0) (#\9)]
      [(ELSE)   (C)]
     )
   )
 )
```

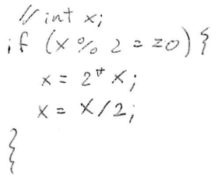
(e) Multistructural



```
(define dp
 (lambda (c)
  (cond ((char<? c #\0) c)
        ((char<? c #\9)
         (integer->char
          (- (char->integer c) 1)))
        ((char=? c #\9) #\0)
        (else     c)
   )))
```

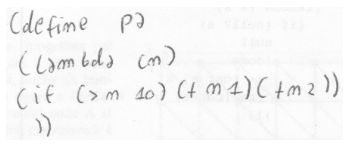
(f) Unistructural

Figure 4: Sample answers at different SOLO levels for imperative (top line) and functional (bottom line) tests.



```
// int x;
if (x % 2 ==0) {
   x = 2* x;
   x = x/2;
}
```

(a) Imperative



```
(define  pd
 ((lambda  (m)
 (if  (> m 10) (+ m 1)(+ m 2))
  ))
```

(b) Functional

Figure 5: Sample answers for pre-structural SOLO level.

rating process based on the criteria outlined in Table 1. As is customary, SOLO means are determined by averaging over the weights assigned to each level [19]: 4=relational, 3=multistructural, 2=unistructural, 1=prestructural, and 0=no justification. Additionally, in the rest of this subsection, we will briefly describe and provide samples for each SOLO level.

Relational answers. Relational justifications, which are correct and accurate, amount to about two thirds of the answers overall for the imperative test and about 55% for the functional test. A couple of examples are shown in Figure 4.a relative to the imperative item (i) and Figure 4.d for the functional item (ii).

Table 3 shows how consistently students appear to work at the relational level, in terms of number of relational justifications.

Multistructural answers. Overall, about 17% of the answers for the imperative test, but only 6% for the functional test, have been rated at this level. A representative example, relative to the imperative test, is shown in Figure 4.b. There, the student has tried to adjust the condition in accordance with the effect of the operation in the original code, without paying attention that it holds not only for the doubles of even numbers but also for every second odd number. In other words, the approach is reasonable, but the task is not completely mastered. A representative example of a multistructural attempt to reverse the functional program (iv) is shown in Figure 4.e, where each individual case has been identified and dealt with appropriately, but the order in which the conditions are evaluated is incorrect.

We classified at this level also trials to reverse the imperative code (i) by changing the assignment and reworking the condition accordingly, while however missing the state overlap.

Unistructural answers. Answers at this SOLO level — overall about 9% for the imperative test and 30% for the functional test — reflect a naive approach in which each component of the conditional construct is edited *independently* of the others. In particular, it includes functional's answers where the students reversed the operations in the *if* branches, but simply copied the original condition, so failing to coordinate *multiple* structures in order to achieve the task at hand. Another recurrent issue for item (ii) of the functional test is shown in 4.f., where the expression in the third clause is reversed, but everything else is left unaffected (contrast this to Figure 4.e).

Other unistructural answers result from attempts to reverse the program by manipulating the conditional almost syntactically, as in the example in Figure 4.c where both the expression in the assignment and the condition are carelessly reversed.

Prestructural answers. Overall, less than 6% of the answers of the imperative cohort, as well as of the *full* functional cohort, have been classified as prestructural. In the imperative test, several such answers exhibit the issue of immediately undoing the change inside the original code, as exemplified in Figure 5.a for item (iv).

A few programs result from some sort of "mechanical" manipulation of the conditional constructs, as in the example of Figure 5.b, relative to the functional item (ii), where the expressions in the two branches are simply swapped. Other prestructural answers report odd explanations or reveal some lack of understanding of the reversibility concept.

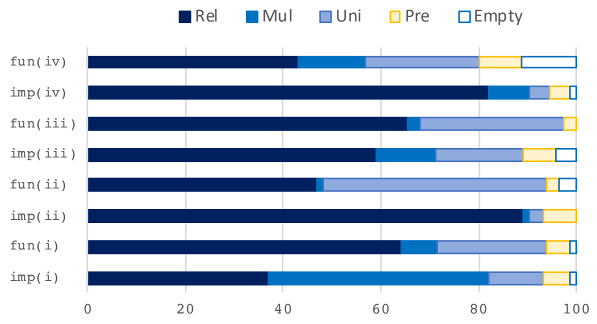


Figure 6: SOLO distribution for each item (i–iv) for either imperative or functional context.

Student's consistency. A different view of the data about students' answers is shown in Table 3. There we can see how the number of correct options and relational justifications are distributed within the observed groups. The dark-gray colored cells indicate consistent answers in that each correct answer is suitably justified. When the number of justifications relative to correct options drops significantly, i.e. 2 out of 3 correct options are not appropriately justified, we could infer the selection arose from partial understanding of the code or even from a random guess. It is worth noting that more that half of students in each cohort could provide at least 3 relational justifications.

Question difficulty. The SOLO means shown in Table 2 reflect the question difficulty for each cohort, while figure 6 shows the percentages of answers at each SOLO level for each item. For the imperative cohort, item(i) shows the lowest rate of relational answers, with most students working at the multistructural level by reversing the code, which will work for all values but 10. The items (ii) and (iv) have the highest SOLO means, with more than 80% of the cohort classified at the relational level. For the functional cohort, providing counterexamples when the code is non-reversible appears to be easier (more than 85% of students that selected "No" did so) compared to working out the correct functional code in the reversible case.

In short, although the distribution of justifications shown in Table 3 are not dissimilar for the two groups, the imperative cohort obtained higher results relative to the items (ii) and (iv), which we attribute to the fact it was possible to reverse the code without editing the conditions as explained in section 3.1. We have investigated this hypothesis by performing a finer-grained analysis of the answers for the students that attempted to reverse item (ii). Table 4 distinguishes the types of treatment of the *if* condition and reports the corresponding frequency: while some students reworked the condition to compensate for the effect of the operation on the variable state or on the returned value, others simply used the given condition without changes in their reversing program. Table 4 will be interpreted and discussed in the next section.

5 DISCUSSION

We proceed now by revisiting the results presented in the previous section in light of the research questions.

Table 3: Number of relational justifications vs. number of correct options per student for each cohort.

	correct options	relational justifications 4	3	2	1	0	total
imperative	4	30.1%	11.0%	4.1%	—	—	45.2%
	3		16.4%	13.7%	2.7%	—	32.9%
	2			11.0%	4.1%	5.5%	20.5%
	1				—	1.4%	1.4%
	total	30.1%	27.4%	28.8%	6.8%	6.8%	100%
functional	4	22.2%	14.8%	6.2%	1.2%	1.2%	45.7%
	3		13.6%	3.7%	7.4%	2.5%	27.2%
	2			4.9%	7.4%	8.6%	21%
	1				—	6.2%	6.2%
	total	22.2%	28.4%	14.8%	16.0%	18.5%	100%

Table 4: Analysis of justifications associated to answers reporting a correct option (*Yes*) for item (ii).

% of answers	**Imperative**	**Functional**
with correct reworked condition	52.1%	57.6%
with unedited condition	39.4%	27.3%
other	8.5%	15.1%
correct code (relational)	91.5%	57.6%

In regards to RQ1, from Table 3 we can see that 45% of students in each cohort chose the right option in all four cases. However, when we look at the quality of their justifications, only 30% for the imperative test, and 22% for the functional test, were able to provide good counterexamples for non reversible items and correct program reversal for the reversible ones. It then appears that it is not so easy to have a comprehensive grasp of the implications of a simple conditional construct, be it imperative or functional.

Recognising that a program based on a conditional can*not* be reversed seems to be more challenging than guessing that it can. This may be explained by the difficulty to identify border computations that would suggest suitable counterexamples.

It is worth observing that the overall SOLO means, across all four items, are 3.41 and 3.03 for the imperative and functional cohorts respectively, which compares well to other studies using SOLO to assess skills of novice programmers; for instance, Sheard et al. [19] classified students' reading ability with short code fragments to swap the values of two variables, and reported a mean score of 2.39. (Note, moreover, that in our studies both the reversibility concept and the task were completely unfamiliar to the students.)

In regards to RQ2, the performances of the functional group vs. the imperative cohort diverge significantly for item (i), both in terms of chosen options as well as of correct justifications. A most likely explanation of this phenomenon brings into play the implicit *else* branch of the imperative conditional statement. Apparently, students tend to consider only the branch they can see and so ignore the overlap. Adding an explicit else as in item (iii) increases the number of correct answers from 49.3% to 80.8% in the imperative case, although the reasoning behind both statements follows a similar thought process. As instructors, we have seen novices writing *else* redundant code such as *x = x*, which may be a cue of their inaccuracy about the final state for the statement in (i).

There is also a strong divergence between the relational answers of the two groups for the reversible items (ii) and (iv), as shown in Table 2. The difference may be partly attributed to the shorter programming experience of the participants in the functional test. However, we presume that it is mostly due to the increased difficulty for items (ii, iv) in the functional test, because of the need to rework the conditions. This explanation is also substantiated by the fine-grained analysis of item (ii) which is shown in Table 4. Indeed, we can observe that the percentage of correctly reworked conditions is similar for both cohorts, in the range 50–60% — as is the the overall percentage of correctly reworked and unedited conditions. Going back to figure 3, the right side diagram shows the value 10 is not a possible final state of *x* for the imperative task; thus, asking *"x > 10"* or *"x > 11"* would provide the same outcome for the reversing code, therefore the unedited condition is correct. It is unclear, though, how many of the approx. 40% of the students who did not modify the condition of item (ii) in the imperative test did so intentionally. Moreover, if more students attempt to edit the condition, as needed in the functional case, we expect a higher rate of incorrectly edited conditions, which is included in the "other" row of Table 4.

Finally, we have already mentioned in the discussion above that many novice programmers are not at ease with coordinating operations and conditions to reverse a program. This is reflected in the number of answers classified as unistructural for each item, which ranges from 3% to 18% in the imperative test and from 22% to 46% in the functional test. Again, performances of the latter cohort are worse due to the greater difficulty of some subtasks. Students at that SOLO level can write code that is syntactically correct but does not achieve the correct result due to poor coordination.

Limitations. As with most exploratory studies, a key limitation is that the investigation involved two cohorts from one institution in consecutive years, thus it remains unclear to what extent the results can be generalised to other student populations. However, based on the findings in [1], where the performances on items (i) and (ii) of the imperative test have been compared for two cohorts from different countries, we can expect that the problems pointed out here are widespread among novice programmers.

Implications for educators. The repeated experiment has identified a recurrent weakness that concerns students' ability to carry out a comprehensive case analysis and to abstract from a *stepwise* explanation of program behaviour, failing "to see the forest for the trees" [11]. This weakness appears under both imperative and functional programming contexts. The additional semester practice does not help to reduce this weakness overall. In fact, the borderline overlap is more likely to be ignored for item (i) of the imperative test due to the implicit *empty* else.

It is then likely that we overestimate students' ability to plan testing and debugging of small programs, tasks they are usually expected to accomplish by the end of a CS1 course. This will suggest that the reversibility tasks considered here could be a helpful addition to CS1 instruction to explain and explore interactions between computation flows in conditional statements.

Using them towards the end of CS1 as a short intervention or revision may also help to improve students' higher-order thinking skills. Besides, we could scaffold this activity for weaker students, for example by giving them a reversible, say imperative, program construct and asking them to code its reversal, then revise with them the two steps required: (1) reverse each variable update and (2) adjust the conditional statement that triggers the update.

Additionally, this set of tasks provides a concrete example of what thinking in a comprehensive ways means compared with other revision tasks, such as multiple choice questions that trace a small program for a single input value and are often approached cursorily. This comprehensive reasoning can also be modelled with related tasks such as asking CS1 students to use the border case analysis to not only decide on reversibility, but also develop and document testing and debugging plans.

6 CONCLUSIONS

To contribute new insights into program comprehension, we have investigated the ability of two first year undergraduate cohorts to analyse and reverse tiny imperative and, respectively, functional programs based on conditional constructs. Their answers have been thoroughly analysed under the lens of the SOLO taxonomy. Overall, considering the task's novelty, students' performance was not below expectations: more than half of the students could provide relational justifications for at least 3 of the 4 conditional examples. This translated on a high SOLO mean (both cohorts have a mean value > 3) compared to related studies, such as [19], that use this framework to evaluate their coding progress.

This analysis has provided the following **insights** about novices' mastery of conditional constructs:

(a) Building a comprehensive view of the implications of simple conditional constructs is a challenging task for novice programmers in either imperative or functional languages. In particular, students do not seem to be careful enough while dealing with border computations.
(b) The lack of an explicit *else* branch — a typical situation in an imperative context — turns out to affect remarkably students' analysis of the code behaviour.
(c) A significant fraction of students appear to face problems in order to master the coordination between conditions and operations in the reversing code.

This type of reversibility exercises could aid instructors to assess progress of novices who already have mastered the "atomic concepts" of syntax and semantics addressed in [13]. Asking for justifications as in this study provides useful insights into their mental models, which could help to early detect latent misconceptions.

We are currently extending our investigation on reversibility to cover different tiny programs, namely simple loops (imperative paradigm) and recursive constructs (functional paradigm), while also looking for opportunities to include other student cohorts. Furthermore, future work will aim at designing related tasks in order to gain more insight into students' mental models and way of reasoning while deciding that a program cannot be reversed or while writing a reversing program.

REFERENCES

[1] Authors. 2018. To be added later. In *Proceedings of an International Conference*.

[2] J. B. Biggs and K. F Collis. 1982. *Evaluating the quality of learning: The SOLO taxonomy*. Academic Press, New York, USA.

[3] R. Bornat, S. Dehnadi, and D. Barton. 2012. Observing Mental Models in Novice Programmers. In *Proc. 24th Annual Workshop of the Psychology of Programming Interest Group*. Article 6, 7 pages.

[4] Ibrahim Cetin. 2015. Student's Understanding of Loops and Nested Loops in Computer Programming: An APOS Theory Perspective. *Canadian Journal of Science, Mathematics and Technology Education* 15, 2 (Feb. 2015), 155–170. https://doi.org/10.1080/14926156.2015.1014075

[5] Yuliya Cherenkova, Daniel Zingaro, and Andrew Petersen. 2014. Identifying Challenging CS1 Concepts in a Large Problem Dataset. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education (SIGCSE '14)*. ACM, New York, USA, 695–700. https://doi.org/10.1145/2538862.2538966

[6] Kevin Cox and David Clark. 1998. The Use of Formative Quizzes for Deep Learning. *Computers & Education* 30, 3-4 (April 1998), 157–167. https://doi.org/10.1016/S0360-1315(97)00054-7

[7] Cruz Izu, Cheryl Pope, and Amali Weerasinghe. 2017. On the Ability to Reason About Program Behaviour: A Think-Aloud Study. In *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '17)*. ACM, New York, USA, 305–310. https://doi.org/10.1145/3059009.3059036

[8] F.J. King, L. Goodson, and F. Rohani. 1998. Higher order thinking skills: Definitions, strategies, assessment. Florida State University – Center for Advancement of Learning and Assessment. (1998). http://www.cala.fsu.edu/files/higher_order_thinking_skills.pdf accessed Nov. 2017.

[9] Raymond Lister. 2007. The Neglected Middle Novice Programmer: Reading and Writing without Abstracting. In *Proceedings of the 20th Conference of the National Advisory Committee on Computing Qualifications (NACCQ'07)*, S. Mann and N. Bridgeman (Eds.). 133–140.

[10] Raymond Lister. 2011. Concrete and other neo-piagetian forms of reasoning in the novice programmer. *Conf. Res. Pract. Inf. Technol. Ser.* 114 (2011), 9–18.

[11] Raymond Lister, Beth Simon, Errol Thompson, Jacqueline L. Whalley, and Christine Prasad. 2006. Not Seeing the Forest for the Trees: Novice Programmers and the SOLO Taxonomy. In *Proceedings of the 11th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ITICSE '06)*. ACM, New York, USA, 118–122. https://doi.org/10.1145/1140124.1140157

[12] Mike Lopez, Jacqueline Whalley, Phil Robbins, and Raymond Lister. 2008. Relationships Between Reading, Tracing and Writing Skills in Introductory Programming. In *Proc. 4th Int. Workshop on Computing Education Research (ICER '08)*. ACM, New York, USA, 101–112. https://doi.org/10.1145/1404520.1404531

[13] Andrew Luxton-Reilly, Brett A. Becker, Yingjun Cao, Roger McDermott, Claudio Mirolo, Andreas Mühling, Andrew Petersen, Kate Sanders, Simon, and Jacqueline Whalley. 2017. Developing Assessments to Determine Mastery of Programming Fundamentals. In *Proceedings of the 2017 ITiCSE Conference on Working Group Reports (ITiCSE-WGR '17)*. ACM, New York, USA, 47–69. https://doi.org/10.1145/3174781.3174784

[14] Philipp Mayring. 2014. *Qualitative content analysis: theoretical foundation, basic procedures and software solution*. Klagenfurt, Austria.

[15] Robert McCartney, Jan Erik Moström, Kate Sanders, and Otto Seppälä. 2004. Questions, Annotations, and Institutions: observations from a study of novice programmers. In *Proceedings of the 4th Koli Calling International Conference on Computing Education Research (Koli Calling '04)*. ACM, New York, USA, 11– 19.

[16] Nancy Pennington. 1987. Comprehension Strategies in Programming. In *Empirical Studies of Programmers: Second Workshop*, Gary M. Olson, Sylvia Sheppard, and Elliot Soloway (Eds.). Ablex Publishing Corp., Norwood, NJ, USA, 100–113.

[17] Jean Piaget. 1999. *The Construction of Reality in the Child*. Routledge. Original edition: La représentation du monde chez l'enfant, F. Alcan, Paris, 1926.

[18] Ian Sanders, Vashti Galpin, and Tina Götschi. 2006. Mental models of recursion revisited. In *Proceedings of the 11th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ITICSE '06)*. ACM, New York, USA, 138–142.

[19] Judy Sheard, Angela Carbone, Raymond Lister, Beth Simon, Errol Thompson, and Jacqueline L. Whalley. 2008. Going SOLO to Assess Novice Programmers. In *Proceedings of the 13th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE '08)*. ACM, New York, USA, 209–213. https://doi.org/10.1145/1384271.1384328

[20] Shuhaida Shuhidan, Margaret Hamilton, and Daryl D'Souza. 2009. A Taxonomic Study of Novice Programming Summative Assessment. In *Proc. 11th Australasian Conf. on Computing Education - Volume 95 (ACE '09)*. Australian Computer Society, Inc., Darlinghurst, Australia, 147–156.

[21] Peter Sutherland. 1999. The application of Piagetian and Neo-Piagetian ideas to further and higher education. *International Journal of Lifelong Education* 18, 4 (1999), 286–294. https://doi.org/10.1080/026013799293702

[22] Donna Teague and Raymond Lister. 2014. Programming: Reading, Writing and Reversing. In *Proceedings of the 2014 Conference on Innovation and Technology in Computer Science Education (ITiCSE '14)*. ACM, New York, USA, 285–290. https://doi.org/10.1145/2591708.2591712

[23] Errol Thompson, Jacqueline Whalley, Raymond Lister, and Beth Simon. 2006. Code Classification as Learning and Assessment Exercise for Novice Programmers. In *Proceedings of the 19th Annual Conference of the National Advisory Committee on Computing Qualifications*, S. Mann and N. Bridgeman (Eds.). NACCQ in cooperation with ACM SIGCSE, 291–298.

[24] Jacqueline Whalley, Tony Clear, Phil Robbins, and Errol Thompson. 2011. Salient elements in novice solutions to code writing problems. *Conferences in Research and Practice in Information Technology Series* 114 (2011), 37–45.