Logical issues in the teaching and learning of proof and proving

Part 2 : Syntax, semantics and proof in CS education

PAT 2023, Val d'Ajol

Viviane Durand-Guerrier
Antoine Meyer

Who am I?

Associate professor at Université Gustave Eiffel
(the university formerly known as Université Paris Est - Marne-la-Vallée)

Laboratoire d'Informatique Gaspard Monge

Sous la co-tutelle de :
CNRS
ÉCOLE DES PONTS PARISTECH
UNIVERSITÉ GUSTAVE EIFFEL

Last few decades

- Engineering degree & PhD in computer science
- Teaching general computer science
  (especially intro to algorithms & programming)
- Research in theoretical computer science
  (logics, automata, infinite graphs, verification...)

Last few years

- Master's degree in didactics of sciences (Univ. Montpellier)
- Some research in CS & math education
  (algorithms & complexity, programming, proof...)

Last few months

A priori analysis of several proof assistants for education
with Bartzia, Beffara & Narboux 👋

(Cf. our session same time tomorrow)

Aim of this session

Attempt to complement Viviane's lectures from a computer science point of view

- based on elementary examples and observations in high school / undergraduate CS curricula
- *not* actual research results

(Disclaimer: not sure about the level of detail...)

Outline

1. Interplay between syntax and semantics when teaching and learning programming
2. Basic types of proofs in computer science education (especially in programming)
3. Concluding remarks on the possible roles of logic, formal proof and "proof technology"

1) Syntax and semantics for beginner programmers

General observation

When learning a new programming language (such as Python in high-school or 1st-year undergrad), *syntax* and *semantics* are often discovered simultaneously

- `x // 2` is a well-formed Python expression, and so is `x // 0`
- `5 // 2` evaluates to `2`
- evaluating `5 // 0` throws an arithmetic error
- `"5" // 2` throws a typing error, etc.

more detail below

In [1]:
```
from dis import dis
dis("x // 0")
```

```
  0           0 RESUME                   0

  1           2 LOAD_NAME                0 (x)
              4 LOAD_CONST               0 (0)
              6 BINARY_OP                2 (//)
             10 RETURN_VALUE
```

In [2]:
```
5 // 2
```

Out[2]: 2

In [6]:
```
5 // 0
```

```
---------------------------------------------------------------------
--------
ZeroDivisionError                         Traceback (most recent ca
ll last)
Cell In [6], line 1
----> 1 5 // 0

ZeroDivisionError: integer division or modulo by zero
```

In [4]:
```
"5" // 2
```

```
---------------------------------------------------------------------
--------
TypeError                                 Traceback (most recent ca
ll last)
Cell In [4], line 1
----> 1 "5" // 2

TypeError: unsupported operand type(s) for //: 'str' and 'int'
```

Syntax

Syntax can be defined *de facto* by the set of texts accepted by the interpreter or compiler

- Underlying models: grammar, syntax tree (abstract or concrete)
- Not always fully formalized and/or documented

Example: excerpt from Python's grammar

```
compound_stmt ::=  if_stmt | while_stmt | for_stmt | ...
if_stmt       ::=  "if" assignment_expression ":" suite
                   ("elif" assignment_expression ":" suite)*
                   ["else" ":" suite]
suite         ::=  stmt_list NEWLINE
                   | NEWLINE INDENT statement+ DEDENT
statement     ::=  stmt_list NEWLINE | compound_stmt
```

Semantics

Semantics can be inferred (or defined *de facto*) from the behaviours of programs at execution

- Underlying model: various flavours of formal semantics
  (deduction rules, sequents...)
- Not always fully formalized and/or documented

Example: typical operational semantics for a while loop in a toy imperative language

$$\frac{\langle b, \sigma\rangle \to \mathbf{false}}{\langle \mathbf{while}\ b\ \mathbf{do}\ c, \sigma\rangle \to \sigma}$$

$$\frac{\langle b, \sigma\rangle \to \mathbf{true} \quad \langle c, \sigma\rangle \to \sigma'' \quad \langle \mathbf{while}\ b\ \mathbf{do}\ c, \sigma''\rangle \to \sigma'}{\langle \mathbf{while}\ b\ \mathbf{do}\ c, \sigma\rangle \to \sigma'}$$

(from Winskel, 1993)

A complex *milieu*

Programs are

- translated (by a compiler or interpreter)
- executed on a machine (abstract or concrete)

"Layered" semantics:
problem ↔ algorithm ↔ program ↔ "translator" ↔ machine ↔ environment

Abstraction may help (but only so far)

Example 1: constructing a model of variables in Python

Aim of this example: reflexion on the need to make (at least part of) the semantics explicit

How to explain the code below?

In [7]:
```
x = 3
```

In [8]:
```
x
```

Out[8]:
```
3
```

In [9]:
```
x = x + 1
```

In [10]:
```
x
```

Out[10]:
```
4
```

In [11]:
```
type(x)
```

Out[11]:
```
int
```

Possible "pragmatics":

- A variable is like a "box" with a name, containing a value
- A variable has a **type** (here `int`) denoting the kind of value it can contain
- Assigning `x + 1` to `x` takes the current value out of box `x`, increments it and puts it back in the box

How to explain the code below?

In [12]:
```
x = 3
y = x
x = x + 1
```

In [13]:
```
x, y
```

Out[13]:
```
(4, 3)
```

Possible pragmatics:

- `x` and `y` are two different boxes,
  therefore changing the content of box `x` has no impact on box `y`
- `y = x` *duplicates* the value contained in box `x` and stores the copy in `y`

How to explain the code below?

In [14]:
```
def f(x):
    x = x + 1
```

In [15]:
```
x = 3
f(x)
```

In [16]:
```
x
```

Out[16]:
```
3
```

Possible pragmatics:

- Parameter `x` in function `f` denotes a different box than the top-level (*global*) variable `x`
- Calling `f(x)` *copies* the value from global box `x` to parameter-box `x`

But :

In [17]:
```
x = 3
x = [3]
x = "3"
```

Contradicts previous pragmatics: `x` "changes type" during the same execution

In [18]:
```
x = [3]
y = x
x[0] = x[0] + 1
```

In [19]:
```
x, y
```

Out[19]:
```
([4], [4])
```

Contradicts previous pragmatics: `x` and `y` not independent.

In [20]:
```
def f(x):
    x[0] = x[0] + 1
```

In [21]:
```
x = [3]
f(x)
```

In [22]:
```
x
```

Out[22]:
```
[4]
```

Contradicts previous pragmatics: global `x` and local `x` not independent.

(one more example below, skip if the point is made)

In [ ]:
```
def null_matrix(n):
    matrix = []
    row = [0] * n
    for _ in range(n):
        matrix.append(row)
    return matrix
```

In [ ]:
```
m = null_matrix(4)
m[0][0] = 1
```

In [ ]:
```
m
```

Contradicts previous pragmatics: rows not independent.

Visual explanation (https://pythontutor.com/visualize.html#code=def%20null_matrix%28n%29%3A%0A%20%20%20%20matrix%20%3D%20%5B%5D%0A%20%20%20%20row%20%3D%20%5B0%5D%20*%20n%0A%20%20%20%20for%20_%20in%20range%28n%29%3A%0A%20%20%20%20%20%20%20%20matrix.append%28row%29%0A%20%20%20%20return%20matrix%0A%20%20%20%20%0Am%20%3D%20null_matrix%284%29%0Am%5B0%5D%5B0%5D%20%3D%201&cumulative=false&curInstr=0&heapPrimitives=true&mode=display&origin=opt-frontend.js&py=3&rawInputLstJSON=%5B%5D&textReferences=false)

Summary of observations

Naive semantics fail to match the experience

Several ingredients missing:

- Dynamic typing
- Object references and aliasing
- Mutability

Revised "Python beginner" semantics:

- Variables are *untyped* labels or *references* on Python objects
- Any object can be designated by several names (aliasing)
- Some objects (such as the number `3`) are *immutable*,
  some (such as the list `[3]`) are *mutable*
- Objects are passed as arguments to functions via their *references*

These examples suggest that:

- *Not* making explicit some aspects of language semantics may hinder learning
- Beginner-oriented semantic models should be consistent with observable behaviour
- **But** revealing the full semantics on week (year?) 1 might not be practical / realistic

Example 2 : what is a list?

Aim of this example: reflexion on the need to build appropriate *conceptions* of data

Possible definition

A *list* is a sequence of elements indexed by natural numbers

Ambiguous:

- type of elements?
- finite or infinite?
- allowing what operations?

Operations

- Tell the length of a (finite) list
- Access or modify the first, last, i_{th} element
- Insert or remove one or several elements at a given position
- Search or count elements with a certain property
- Concatenate, iterate, reverse, shuffle, sample a list
  (or if that's your thing: map it, fold it, zip it...)
- **Sort** a list, perform **binary search** on a sorted list, etc.

Note: depending on context, some operations may be considered elementary while others can be derived

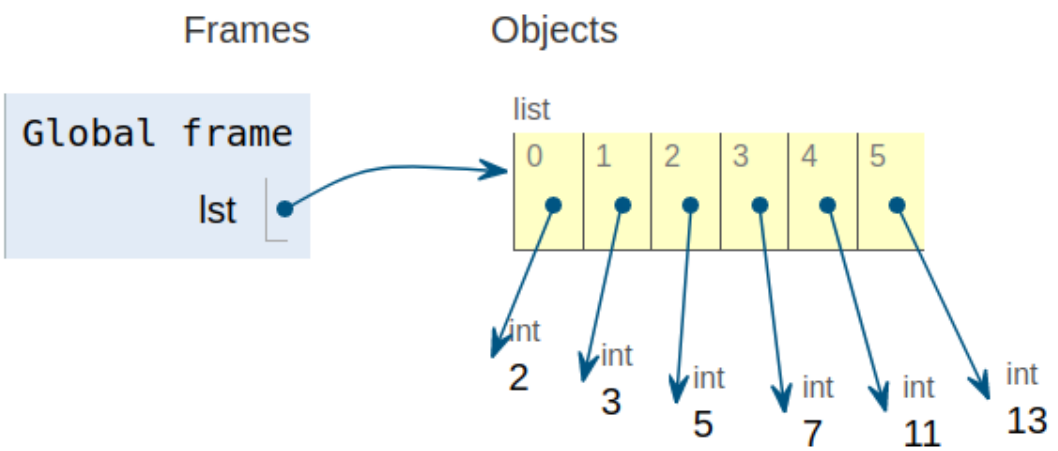
Are these lists ?

A "mathematical" sequence

$u = (2, 3, 5, 7, 11, 13)$

A native Python list

In [23]:
```
small_primes = [2, 3, 5, 7, 11, 13]
```



(image stolen from http://www.pythontutor.com (http://www.pythontutor.com))

(examples of native list operations below)

In [ ]:
```
smallest_prime = small_primes[0]
```

In [ ]:
```
smallest_prime
```

In [ ]:
```
small_primes.append(20)  # insertion at the end
```

In [ ]:
```
small_primes
```

In [ ]:
```
small_primes[6] = 19  # cell modification
```

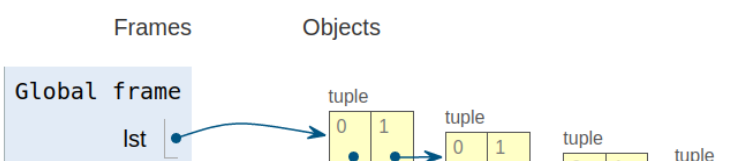
In [ ]:
```
small_primes
```

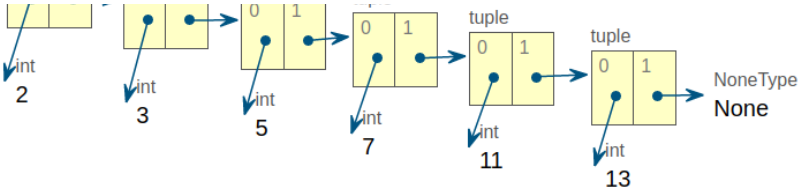
In [ ]:
```
small_primes.insert(6, 17)  # insertion in sixth position
```

In [ ]:
```
small_primes
```

A "recursive" list with a (head, tail) representation

In [24]:
```
small_primes = (2, (3, (5, (7, (11, (13, None))))))
```





(examples of operations below)

In [ ]:
```
first_prime = small_primes[0]  # head of the list
```

In [ ]:
```
first_prime
```

In [ ]:
```
small_odd_primes = small_primes[1]  # tail of the list
```

In [ ]:
```
small_odd_primes
```

In [ ]:
```
third_prime = small_primes[1][1][0]
```

In [ ]:
```
third_prime
```

A curiosity below (maybe skip?)

An infinite list of integers:

```
let naturals = [1..]   -- valid in Haskell
```

The first 10 terms of the Fibonacci sequence:

```
> fibo = 0 : zipWith (+) fibo (1:fibo)

> take 10 fibo
[0,1,1,2,3,5,8,13,21,34]
```

An example of list usage: insertion sorting

(skip?)

Algorithm (basic idea)

To sort some list L of pairwise comparable elements:

- Initialize a new list R
- Consider each element e of L in turn
  - *Insert* e in R so as to leave R sorted

Implementation using mutable array-based lists

In [28]:
```
def insertion_sort(lst):
    for i in range(1, len(lst)):
        # place element i with respect to previous elements
        insertion(lst, i)
```

In [25]:
```
def swap(lst, i, j):
    lst[i], lst[j] = lst[j], lst[i]
```

In [26]:
```
def insertion(lst, i):
    # move new element to the "left" until it is well-placed
    while i > 0 and lst[i-1] > lst[i]:
        swap(lst, i-1, i)  # can be improved
        i = i - 1
```

In [29]:
```
lst = [5, 19, 9, 6, 13, 15, 18, 14, 0, 17]
insertion_sort(lst)
lst
```

Out[29]:
```
[0, 5, 6, 9, 13, 14, 15, 17, 18, 19]
```

Implementation using immutable recursive lists

In [30]:
```
def insertion_sort(lst):
    if lst == None:
        return lst
    else:
        # sort the tail
        tmp = insertion_sort(lst[1])
        # re-insert the head
        return insertion(lst[0], tmp)
```

In [31]:
```
def insertion(e, lst):
    if lst == None or lst[0] > e:
        # insert new element at head
        return (e, lst)
    else:
        # insert in tail and restore head
        return (lst[0], insertion(e, lst[1]))
```

In [32]:
```
lst = (5, (19, (9, (6, (13, (15, (18, (14, (0, (17, None))))))))))
insertion_sort(lst)
```

Out[32]:
```
(0, (5, (6, (9, (13, (14, (15, (17, (18, (19, None))))))))))
```

Discussion

Characteristics of the array-based implementation:

- Accesses elements using indices
- Exploits mutability and function side-effects

Characteristics of the recursive-list implementation:

- Operates only on the head of lists
- Recursive reasoning, almost equational
- Does not modify the original list

How similar are these two proposals?

- Do they implement the same algorithm?
- Do they correspond to the same idea of *what a list is*?
- Are they comparably efficient?

Building a mental model of data

Hypothesis: it is fundamental for learners to build a clear "mental image" of the way data is organized

- Helps to understand the semantics of data operations *(mutability!)*
- Large impact on problem solving, algorithm design and programming patterns
- Required to understand statements about complexity

Complexity of a few operations on recursive and array-based lists

operation	recursive lists	array-based lists
length	$O(n)$	$O(1)$
insert or remove first	$O(1)$	$O(n)$
insert or remove last	$O(n)$	$O(1) \leftarrow$ *hum...*
search sorted list	$O(n)$	$O(\log n)$

This is almost impossible to discuss with students without a clear view of what each type of list means...

Remark: Implementation details may sometimes be "abstracted away"
(notion of *abstract data type*)

... with what effects on beginner's learning?

Lists in official high school syllabus (NSI, France)

Unfortunately, knowing what kind of lists are consideres is not always obvious (even for teachers, especially under-trained ones):

> The dynamic aspect of Python arrays is not mentioned. Python makes no distinction between lists and arrays.
> (Programmes NSI 1ère, 2019, "Indexed arrays" entry)

> Lists do not exist natively in Python.
> (Programmes NSI Tle, 2019, "Lists, stacks, queues: linear structures" entry)

Remember, the native Python type is called... `list`

Conclusion (part 1) : didactical issues regarding syntax and semantics

Hoc & Nguyen-Xuan (1990) write (my emphasis):

> "In teaching programming, the emphasis stressed on the means of expression can lead to an **overestimation of the learning of programming language syntax** within the triad determining human-computer interaction: external task structure, language syntax, and language semantics (Moran, 1981)."

> "Beginners learn to program by building programs. Hence they learn by problem- solving (...) by doing and by analogy."

When teaching programming to beginners (or a new language to non-beginners), several didactical issues arise:

- How can one help learners grasp both syntactic and semantic aspects?
- What *epistemologic* or *didactic obstacles* can be identified?
- What "tentative semantics", or abstractions?
- What impact of language (or *paradigm*) characteristics?
- What impact of these choices on other concepts or topics in CS education?
- *(Unrelated?) What impacts on proficiency at learning PAs?* 😇

2) Aspects of proof in algorithms and programming education

Disclaimer:

- Examples voluntarily taken at high-school / *non-selective* undergraduate levels
- Focus on basic algorithm and program analysis, geared towards "explanation"
- **No** focus on theoretical computer science or other higher-level topics

Three main types of proofs regarding algorithms or programs:

termination, *correctness*, *complexity*

Termination

Termination is the property of an algorithm or program to only perform a finite number of elementary computation steps on any "suitable" instance

Points of attention:

- what is an elementary step?
- how can we specify which instances are suitable?

Usual proof technique: infinite descent w.r.t. some *well-founded* relation (contradiction)

(examples below)

Example: termination of the recursive `insertion` function

In [33]:
```
def insertion(e, lst):
    if lst == None or lst[0] > e:
        return (e, lst)
    else:
        return (lst[0], insertion(e, lst[1]))
```

The function:

- stops if the list is empty
- calls itself on a strictly shorter list otherwise

Example: termination of the iterative `insertion` function

In [34]:
```
def insertion(lst, i):
    while i > 0 and lst[i-1] > lst[i]:
        swap(lst, i-1, i)  # can be improved
        i = i - 1
```

In the function's body:

- the value of `i` starts above 0
- `i` decreases strictly at each iteration
- the loop stops if `i` reaches 0

Correctness (or *partial* correctness)

An algorithm or program is correct if, *assuming* it stops on a given instance, the result is consistent with the specification regarding this instance

Points of attention:

- does not suppose termination

- how is the specification expressed?

Usual proof technique: mathematical induction (classical, complete, structural...)
Note: quite natural on purely functional programs

(examples below)

Example: recursive `insertion_sort`

In [35]:
```
def insertion_sort(lst):
    if lst == None:
        return lst
    else:
        return insertion(lst[0], insertion_sort(lst[1]))
```

Specification:

- `insertion` is correct if, *assuming* `lst` is sorted, `insertion(e, lst)` returns a *sorted permutation* of the list `(e, lst)`
- `insertion_sort` is correct if `insertion_sort(lst)` returns a *sorted permutation* of `lst`

(Of course, one needs to define "being sorted" and "being a permutation of...")

Assuming the auxiliary function `insertion` is correct (similar proof), and `lst` is the parameter, we proceed by induction on the length of `lst` :

- if `lst` empty then the result is the empty list (a sorted permutation of `lst`)
- assuming the function is correct on lists shorter than `lst`
  - `insertion_sort(lst[1])` is a sorted permutation of the tail of `lst` (by induction hypothesis)
  - `insertion(lst[0], insertion_sort(lst[1]))` returns the expected result (by assumption on `insertion`)

Formal correctness proof for imperative programs

(Much) more complicated than on functional programs

Possible technique: using Hoare logic

- annotation of program points with predicates over states
- deduction rules (relying on the language's semantics) to ensure coherence
- implicitly embeds *structural induction* on the program's syntax

Hoare triple: judgment of the form

$$\{A\}\ S\ \{B\}$$

with

- A, B : predicates over execution states (values of variables...)

Semantics of a Hoare triple:

$$\{A\}\ S\ \{B\}$$

> Assuming
>
> - predicate A holds in some state σ,
> - S terminates from state σ ending in state σ',
>
> predicate B must hold in σ'

Remarks:

- In $\{A\}\ S\ \{B\}$, A and B are called *pre-* and *post-condition*
- Trivially true for any B when A is invalid
- Trivially true when S does not terminate
- Refers to the syntax *and* semantics of programs!

Example rule: sequence

$$\frac{\{A\}\ S_1\ \{B\}\ ,\ \ \{B\}\ S_2\ \{C\}}{\{A\}\ S_1; S_2 \{C\}}$$

(from Gribomont et al., 2000)

Example rule: conditional

$$\frac{\{A \wedge B\}\ S_1\ \{C\}\ ,\ \ \{A \wedge \neg B\}\ S_2\ \{C\}}{\{A\}\ \text{if } B \text{ then } S_1 \text{ else } S_2\ \{C\}}$$

Example rule: while loop

$$\frac{\{I \wedge B\}\ S\ \{I\}}{\{I\}\ \text{while } B \text{ do } S\ \{I \wedge \neg B\}}$$

Predicate I is called an *invariant* property of the loop

Example rule: consequence rule

$$\frac{\models A \Rightarrow B\,,\ \{B\}\,S\,\{C\}\,,\ \models C \Rightarrow D}{\{A\}\,S\,\{D\}}$$

Notes:

- Proving implications in the underlying logic is not (at all) guaranteed to be easy *(or even decidable...)*
- "Inventing" appropriate intermediate predicates is hard to systematize

Informal correctness arguments for imperative programs

Possible technique: exploit the intuitive ideas of

- pre-condition and post-condition
- loop invariant

without logic formalization or explicit deduction rules

Rule of thumb (simplified): if a property is...

- true right before a loop condition is first evaluated
- preserved by an execution of the loop body

... then it is also true once the loop has exited (if it ever does)

Possible tools:

- Manual or tool-generated step-by-step execution tables on "significant" examples
- Graphic depiction of invariant on an abstract instance (e.g. a list) *(not detailed here)*

Example: correctness of iterative `insertion` (informal)

In [36]:
```
def insertion(lst, i):
    j = i
    while j > 0 and lst[j-1] > lst[j]:
        swap(lst, j-1, j)
        j = j - 1
```

In [ ]:
```
def insertion(lst, i):
    j = i
    while j > 0 and lst[j-1] > lst[j]:
        swap(lst, j-1, j)
        j = j - 1
```

Pre-condition (A) : Elements between `0` and `i-1` are sorted

Loop condition (B) : `j > 0 and lst[j-1] > lst[j]`

In [ ]:
```
def insertion(lst, i):
    j = i
    while j > 0 and lst[j-1] > lst[j]:
        swap(lst, j-1, j)
        j = j - 1
```

Invariant: conjunction of

- (I_1) : Elements between indices `j` and `i` are sorted
- (I_2) : Elements between indices `0` and `i` (**excluding** `j`) are sorted

Execution table for `insertion([2, 5, 6, 4], 3)` :

♯	iteration	statement	list portion	j	B	I_1	I_2
1	∅	∅	[2,5,6,4]	∅	✓	∅	∅
2	∅	j=i	[2,5,6,4]	3	✓	✓	✓
4	1	swap	[2,5,4,6]	3	✗	✓	✗
5	1	j=j-1	[2,5,4,6]	2	✓	✓	✓
4	2	swap	[2,4,5,6]	2	✗	✓	✗
5	2	j=j-1	[2,4,5,6]	1	✗	✓	✓

Same execution, keeping only relevant lines:

♯	iteration	statement	list portion	j	B	I_1	I_2
2	∅	j=i	[2,5,6,4]	3	✓	✓	✓
5	1	j=j-1	[2,5,4,6]	2	✓	✓	✓
5	2	j=j-1	[2,4,5,6]	1	✗	✓	✓

This shows that $I_1 \wedge I_2$ is indeed an invariant of the loop *on this instance*.
If this were true on all instances (it is!) then at the end of the loop we would get:

Post-condition ($C = I_1 \wedge I_2 \wedge \neg B$) : Elements between `0` and `i` are sorted

Note: column `j` illustrates the notion of *variant*, useful to show termination

Complexity

Given some measure of instances' size, estimate the number of elementary steps performed or "units of memory" used on instances of size n

- Requires a computation model
  (notion of *elementary step*, memory model...)
- Often *asymptotic* upper or lower bounds on *worst-case* complexity

→ Not today... 😅

Theorem provers for program analysis?

Several possibilities for formal computer-assisted correctness proofs
(disclaimer: not a specialist here...)

Example 1: direct inductive proofs on Coq functions

(available demo : Coq proof of insertion sort on recursive lists, from Appel, 2020)

Obstacles :

- possible difficulties with functional programming language
- steep learning curve to write and understand Coq proofs

Example 2: Coq formalization of Hoare logic

(available demo: example formalization of Hoare rules, from Pierce et al., 2020)

Obstacles:

- formalization of the target language's semantics required
- a lot of boilerplate (preparatory) code
- rather complex (conceptually and technically)

Example 3: automatic verification of Hoare logic annotations using Why3

(available demo: termination and correctness proof of imperative insertion sort)

Advantages: automatic once annotations are done!
Obstacles:

- not very explanatory (especially when it doesn't work)
- difficult to write correct annotations

(alternative: discharge proofs to Coq!)

3) Concluding remarks

Several aspects of the learning of algorithms and programming clearly rely on "logico-mathematical" activities

- Syntax and semantics of computer programs *(reading **and** writing)*
- Structured problem resolution, anticipation / planning, evaluation
- Specification, error-finding, testing
- Correctness proofs, estimation of complexity

Obstacle: many mathematical ingredients required for formalisation not readily available

- Familiarity with proof in general (including its *necessity*)
- Induction principles (beyond simple induction on $\mathbb{N}$)
- Inductive definition of sets and functions
- Formal logic, deduction systems, notations
- Well-foundedness, etc.

Long(ish)-term research questions in undergraduate algo-prog education

- How can one encourage a reflective mindset regarding program correctness in students?
- What possible (reasonable?) uses for formal logic and proof?
- What tools and didactic situations to leverage the power of ATP and ITP while remaining accessible to beginners?
  (And: is it such a good idea anyway?)

Partial bibliography

Appel (2020). *Verified functional algorithms* (Software Foundations, Vol. 3). Electronic textbook.

Gribomont, Ribbens & Wolper (2000). Logique, automates, informatique. In *Logique en Perspective: Mélanges Offerts à Paul Gochet* (pp. 545–577). Ousia.

Hoc & Nguyen-Xuan (1990). Language Semantics, Mental Models and Analogy. In *Psychology of Programming* (pp. 139–156). Academic Press.

Pierce et al. (2020). *Programming language foundations* (Software Foundations, Vol. 2). Electronic textbook.

Winskel (1993). *The Formal Semantics of Programming Languages: An Introduction*. MIT Press.