Adapting the Programming Language to a Teacher's Didactic Approach

Jesse Hoobergs, KU Leuven

jesse.hoobergs@kuleuven.be

Abstract

When a teacher designs a programming course, they devise the didactic approach that seems most appropriate for the goals and target audience of their course. To implement their course, they will need to choose a programming language. I argue that regardless of the chosen programming language, the implementation of the didactic approach is hampered by the chosen language. In my PhD, I am creating a Programming Education Runtime System that allows teachers to easily create educational languages specifically tailored to their didactic approach. A teacher can create such educational languages by selecting the needed programming language constructs. A construct can be obtained from a library of reusable implementations, or it can be implemented by the teacher themself when a needed construct is not yet available.

1 Short Introduction

I started my PhD in February 2023 at the PL group of Tom Schrijvers at the KU Leuven. Felienne Hermans from the VU Amsterdam is my co-supervisor. The topic of my PhD originated from my experiences in teaching programming to students in both secondary education and higher education. The main goal of my PhD is to allow teachers to change the language to fit their didactic approach instead of them needing to adapt their approach to fit the language.

2 Theoretical Model

A *notional machine* refers to the machine that you are learning to control when learning to program. The term was first used by Du Boulay [3]. It is important to note that each programming language has different features and semantics and therefore a different notional machine. Previous research has implied that it is important to show the appropriate notional machine to novices early on [10]. Appropriate means that the presented notional machine contains the part of the language that can be understood by the novice at that point in time. The presented notional machine will change throughout a course when new programming constructs are explained or known ones are refined. The specific syntax of the language is not part of the notional machine of the programming language. The notional machine of a language and its block-based counterpart, are identical.

In my research, I am currently trying to define two concepts related to the notional machine concept: *notional machine discrepancy* and *programming language inadequacy*. Before explaining these concepts, it is important to stress the distinction between (1) the notional machine of a language and (2) the presented notional machine. The first concept describes a machine corresponding to the entire programming language, while the second concept describes a machine that is explained to students at a *particular point in time*. To make this distinction clear, I will use the following terms: *language machine* and *presented machine*. The presented machine will always be smaller than the language machine and might even be inconsistent with it.

The *notional machine discrepancy* is the difference between the *presented machine* and the *language machine*. It can be seen as the difference between the language that you want to explain at a particular point in time and what your language actually is. To put it more strongly, it is the difference between the machine that you want at that point in time and the machine that you have got. This discrepancy

can lead to issues in understanding error messages as these are written in the context of the *language machine*. Error messages are known to be unreadable by novice programmers [10, 11]. Other issues might arise when a novice–maybe accidentally–uses a feature in a way that is not contained within the presented notional machine. It is impossible for the novice to understand the execution of that program within the presented notional machine. Put differently, the *notional machine discrepancy* is about what a novice can understand at a particular moment and about the fact that the language allows things that the novice cannot yet understand at that moment. A novice cannot make the distinction between (1) not understanding code because they did not fully grasp the presented machine or (2) not understanding code because it cannot be understood within the presented machine.

A *programming language inadequacy* occurs when the language makes it hard to explain or use a concept. Making it hard means that the language imposes you to explain additional concepts that you do not want to explain (yet). One example is using a media computation approach [8] in Python. Python does not have a built-in image class, so you need to explain how to import your image library. Another example in Python is using a `for x in range(N)` construct where the concept of default parameter values is actually used. It might didactically be better to only allow the explicit `for x in range(0, N)` when novices first learn to program.

Each *programming language inadequacy* hampers the didactic approach of a teacher because a teacher has to remedy the inadequacy. Currently, I see the following three options from which a teacher can choose to achieve this:

1. Tell the novice to write certain *magic characters* in the source code and tell them that they cannot understand them but should use them.
2. Explain the concept together with the additional concepts.
3. Change the order of concept introduction.

Option 1 will introduce *notional machine discrepancies* as the novices will use constructs that are not in the *presented machine*. Option 2 will probably lead to a very high cognitive load for the novice, which will hamper learning. The last option seems a reasonable approach, but it is not. When a teacher chooses that option, they are changing their didactic approach to match the language. The language makes them adapt their didactic approach to an approach that they would otherwise never use.

A fourth option would be to look for another programming language that does not suffer from this specific *programming language inadequacy*. However, this programming language will in turn have other *programming language inadequacies* that have to be remedied.

3 Related Work

Prior work on mini languages, sub-languages, educational programming languages and teaching languages can be interpreted as reducing the problems of *notional machine discrepancies* and *programming language inadequacies*. Mini languages like Scratch [13] and Logo reduce the *programming language inadequacy* for a specific didactic approach. However, using Scratch to explain concepts that are not part of Scratch, is impossible. Scratch might also suffer a lot from *notional machine discrepancies* as all blocks are always available to novices which makes it easy to use a block that is not yet explained or cannot yet be understood.

Sub-languages like MiniJava [14] and ProfessorJ [7] contain only a subset of the features of another language (in this case Java). Reducing the amount of features might reduce *notional machine discrepancies* for certain didactic approaches. However, they might also increase the *programming language inadequacy* depending on which subset is chosen. These languages will fit a specific didactic approach but are not adaptable to other approaches.

Educational programming languages like Grace [1], Pyret [2], MuLE [4], Quorum [16] and Hedy [9] are full-fledged standalone programming languages. Most of them will have a rather small *programming language inadequacy* for some didactic approaches. Many of these languages contain, for example, data

types to work with images. They might, however, still suffer from quite some *notional machine discrepancies*. Some of these languages (like Hedy [9] and Grace [1]) remedy this by having a gradual approach with different language levels or dialects. These languages are created with a particular didactic approach in mind and adapting them to your own didactic approach is hard. The most well known example of languages tailored to a particular didactic approach are the teaching languages used in *How to design programs* [5].

Prior work on misconceptions can also be useful, as misconceptions could arise from *notional machine discrepancies*. I have looked at the list of misconceptions from Sorva's dissertation [15] and found a couple of misconceptions that could stem from a *notional machine discrepancy*. Unfortunately, it is hard to be sure that the mentioned misconceptions stem from a *notional machine discrepancy* as such a discrepancy inherently contains a time aspect about which machine has been presented and this information is not available in most papers about misconceptions.

4 My solution

All existing languages for education have one common property: they are created with a particular didactic approach in mind. Adapting these languages to a different approach–or even a slightly different approach–is infeasible for most teachers and computing science education researchers. In my research I am creating a Programming Education Runtime System (PERS) that allows the creation of educational programming languages by combining features and programming language constructs.

After devising their didactic approach, teachers can create the language (or languages) for their course by selecting the features from the library of features in the PERS. When a feature / programming language construct is not yet available, they can add it as a reusable component to the PERS. The modular approach of the system allows teachers to easily create gradually changing languages. The second language can be created as an extension of the first language with some new features and the removal of some old features. A reasonable approach would be to create a different language for each lesson or for each chapter or section of a book. These languages will have no *notional machine discrepancy* and no *programming language inadequacy* as they are tailored to the didactic approach and lesson plan.

The modular approach of the system yields the following advantages:

- Code can be reused between different teaching languages.
- Teachers can easily make small tweaks to languages of other teachers: e.g. changing the order of concept introduction.
- Teachers can easily see which features and constructs are added and removed throughout the different levels. This gives the teacher an overview of which notional machine should be presented to students throughout the learning process.

The current version of the PERS has been used to reimplement Hedy [9]. This implementation has many benefits over the existing implementation:

1. executing the language in the browser works out-of-the-box,
2. no notional machine discrepancy,
3. changing the order of concept introduction is trivial,
4. improved debugging support due to step-by-step execution with source location,
5. explicit interaction with the web platform due to the use of algebraic effects [12] in the PERS,
6. creating a Blockly [6] block-based version works out-of-the-box.

The current version of the PERS does, however, also have some shortcomings:

1. the parsing of the modular languages still has to be done manually
2. features cannot yet be removed when extending a language.

5 Next steps

The definitions and naming of the *notional machine discrepancy* and *programming language inadequacy* concepts are still subject to change. I will undertake the following actions to improve and validate these concepts:

- Look at more misconceptions that are reported in literature and determine whether a *notional machine discrepancy* might be the underlying cause.
- Take an in depth look at mini languages, sub-languages and educational programming languages to see how their language design decisions can be explained by the two concepts.
- Observe students in a CS1 course to see whether the *notional machine discrepancy* does lead to issues.
- Interview course developers to get a clear view of the impact of *programming language inadequacy* on their didactic approach.

The PERS system needs to be improved in a number of ways:

- The parsing of the modular languages needs to be implemented.
- The library of implemented features should be extended.
- Allow removing features from the language when extending a language.
- Create a UI system for teachers to create the languages instead of needing to write Rust code.

This second item will be done by implementing some example teaching languages in the system for different didactic approaches to teach programming in Python. One of these approaches will be used in a course text for computer science in Flemish secondary education.

After improving the PERS, it will be evaluated by letting course developers and teachers use the system.

References

[1] Andrew P Black, Kim B Bruce, Michael Homer, James Noble, Amy Ruskin, and Richard Yannow. Seeking grace: a new object-oriented language for novices. In *Proceeding of the 44th ACM technical symposium on Computer science education*, pages 129–134, 2013.

[2] The Pyret Crew. The pyret programming language.

[3] Benedict Du Boulay. Some difficulties of learning to program. *Journal of Educational Computing Research*, 2(1):57–73, 1986.

[4] Nikita Dümmel, Bernhard Westfechtel, and Matthias Ehmann. Mule–a multiparadigm language for education-the procedural sublanguage. In *EDUCON 2020*, pages 392–401. IEEE, 2020.

[5] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. *How to design programs: an introduction to programming and computing*. MIT Press, 2018.

[6] Neil Fraser. Ten things we've learned from blockly. In *2015 IEEE blocks and beyond workshop (blocks and beyond)*, pages 49–50. IEEE, 2015.

[7] Kathryn E. Gray and Matthew Flatt. Professorj: A gradual introduction to java through language levels. In *Companion of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2003.

[8] Mark Guzdial. A media computation course for non-majors. In *Proceedings of the 8th annual conference on Innovation and technology in computer science education*, pages 104–108, 2003.

[9] Felienne Hermans. Hedy: a gradual language for programming education. In *Proceedings of the 2020 ACM conference on international computing education research*, pages 259–270, 2020.

[10] Shriram Krishnamurthi and Kathi Fisler. 13 programming paradigms and beyond. *The Cambridge handbook of computing education research*, 2019.

[11] Samim Mirhosseini, Austin Z Henley, and Chris Parnin. What is your biggest pain point? an investigation of cs instructor obstacles, workarounds, and desires. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*, pages 291–297, 2023.

[12] Gordon D. Plotkin and Matija Pretnar. Handlers of algebraic effects. In Giuseppe Castagna, editor, *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, volume 5502 of *Lecture Notes in Computer Science*, pages 80–94. Springer, 2009.

[13] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, et al. Scratch: programming for all. *Communications of the ACM*, 52(11):60–67, 2009.

[14] Eric Roberts. An overview of minijava. *SIGCSE Bull.*, feb 2001.

[15] Juha Sorva et al. *Visual program simulation in introductory programming education.* Aalto University, 2012.

[16] Andreas Stefik and Richard Ladner. The quorum programming language. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*, pages 641–641, 2017.