RESEARCH-ARTICLE

Lambdulus: teaching lambda calculus practically

JAN SLIACKY, Czech Technical University in Prague, Prague, Czech Republic

PETR MAJ, Czech Technical University in Prague, Prague, Czech Republic

Published: 25 October 2019

Citation in BibTeX format

SPLASH '19: 2019 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity
October 25, 2019
Athens, Greece

Conference Sponsors:
SIGPLAN

Lambdulus: Teaching Lambda Calculus Practically

Jan Sliacky
Faculty of Informatics
Czech Technical University
Prague, Czech Republic
sliacjan@fit.cvut.cz

Petr Maj
Faculty of Informatics
Czech Technical University
Prague, Czech Republic
majpetr@fit.cvut.cz

Abstract

λ calculus is a great formal introduction to functional programming. However, its abstract nature poses a challenge for many students as they struggle both with the unfamiliarity of functional programming and with the high abstraction and minimalism of λ calculus. As a result, functional programming classes are often delayed to older students who are hoped to be better prepared for appreciating its qualities, as was the case at the authors' university as well. 3 years ago, as part of a redesigned curriculum around programming languages education, we have decided to introduce students to various programming paradigms much sooner than before. In doing so, we faced the problem of explaining very theoretical foundational concepts to very young students in a very short time. We have monitored the achievements and shortcomings of the new course over the past years and as a result of our findings have developed Lambdulus, an interactive and visual evaluator of λ calculus expressions that encourages students to explore the mechanisms of λ calculus by treating it not as a theoretical concept, but as a programming language in its own right.

CCS Concepts • **Software and its engineering → General programming languages**; • **Social and professional topics** → *History of programming languages.*

Keywords λ calculus, programming languages curriculum, educational tools

ACM Reference Format:
Jan Sliacky and Petr Maj. 2019. Lambdulus: Teaching Lambda Calculus Practically. In *Proceedings of the 2019 ACM SIGPLAN SPLASH-E Symposium (SPLASH-E '19), October 25, 2019, Athens, Greece.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3358711.3361629

SPLASH-E '19, October 25, 2019, Athens, Greece

ACM ISBN 978-1-4503-6989-3/19/10…$15.00
https://doi.org/10.1145/3358711.3361629

1 Introduction

The Faculty of Informatics at the Czech Technical University (FIT CTU) has very strong hardware and embedded systems curriculum, which has influenced the way we introduce students to programming. All students are introduced to C in the very first term via the Programming and Algorithms course. This is immediately followed by the second part in the next term, introducing C++ and with it informally Object Oriented Programming and more complicated algorithms and data structures.

Although a plethora of courses on other programming languages of different paradigms were offered to the undergraduate students, none of them was compulsory or concerned itself with the fundamentals of computer programming. The situation was slightly better for graduate master's students where a compulsory course on functional and logical programming existed.

To improve the standing of the undergraduate curriculum with respect to more fundamental concepts in programming languages, first an Object Oriented Programming course has been added and finally, three years ago, the graduate course on functional and logic programming has been renamed to Programming Paradigms (PPA) and integrated into the third term of the undergraduate program.

However, when a course moves from 5th to 2nd year, it cannot simply be renamed. The undergraduate course has a much wider audience from different computer science majors ranging from theoretical computer science to web technologies. Care must be taken to ensure that most of the students will understand not just the concepts presented, but also their application and usefulness. Furthermore, most of the knowledge the graduate course built upon (such as the theory of computation, pre-existing knowledge of functional programming style, etc.) is out of scope for an average second-year student. On the other hand, less knowledge and experience also means fewer habits (good and bad), which offers us the opportunity to not just teach the students functional programming, but to teach them to program functionally.

Apart from the new name, the course had to reflect all this in its syllabus. But even more important question was in what way to teach the more theoretical concepts, namely the λ calculus [1]. Since the course must introduce λ calculus, Lisp and Prolog in the span of a single term, we could not afford to spend the time on building the theoretical background necessary for understanding and appreciating the

value of λ calculus. Instead, we have decided to leverage the fact that our students have so far experienced C and C++ languages, both of considerable power, but even greater complexity and show them λ calculus in a different light. Our idea was to "contrast" the λ calculus to the languages our students are guaranteed to know. To do so effectively we had to approach the λ calculus not as an abstract concept, but as a programming language in its own right.

This realization has shaped the entire course as it propagated to other languages and paradigms introduced in the course as well which we discuss in section 2. To validate our ideas we have surveyed past students of the course and we discuss the outcomes of these surveys in section 2.2. One of the things we have realized from the very beginning was that simply changing the narrative of the lectures is not enough. To present λ calculus as a programming language, we have to allow our students to use it as such - specifically to provide them with tools that make such use easy enough. To this end, we have always employed the use of λ calculus evaluators. However, our surveys have demonstrated that existing solutions were not giving λ calculus the treatment we required and were rarely used by the students. To mitigate, we have developed Lambdulus, a λ calculus expression reducer that treats λ calculus as much as a programming language as possible. We present a detailed description of this tool, its design principles and tight integration with our course in section 3.

We hope that both our approach and our tool can be useful for similar courses and can be extended beyond simple λ calculus where it is currently used. Section 4 discusses future improvements, after which we conclude.

1.1 List of Contributions

The contributions of our work are twofold:

- description of our approach to teach formal λ calculus not as a purely theoretical concept, but as a real programming language. Using this approach we have found that students can be introduced to λ calculus much earlier than before, which helps them better understand and *use* functional programming
- Lambdulus, an advanced λ calculus evaluator designed to support the course by providing the students with a programming language-like interface to λ calculus complete with debugging.

2 Course Description and Evaluation

The PPA course[1] is a 4 ECTS credits course running for 12 weeks in the winter term of the second year of the undergraduate programme at FIT CTU. The course is compulsory for a range of undergraduate majors and over the past three years attracted roughly 150 students per year, making it a medium-sized course at FIT. Each week a 90 minutes lecture for all students precedes 90 minutes long tutorials where students split into smaller groups (of 20 students at most) and engage in exercises supervised by either the lecturers or experienced TAs, a format typical for the CTU.

To fulfill the course, students must pass 2 mid-term tests and an exam. The exam consists of a written test, which, contrary to most courses of this size at the faculty, is always followed by an oral part where the students are required to put the gained knowledge in context. Students are encouraged to gain extra points by implementing a programming project from selected topics, without these points the best mark attainable is B (on the standard A - F (Failure) scale).

Covered Topics The lectures of the PPA course are tightly packed as they must cover imperative, functional and logic programming paradigms. First, the imperative programming style is summarized and basic execution techniques for stateful languages are presented (1 week). This is followed by the introduction of λ calculus, which is the focus of this paper and its transition into Lisp (7 weeks). Finally, logic programming is introduced and demonstrated on Prolog language (4 weeks). To make the theoretical aspects less intimidating, a great deal is placed upon "concreteness" of the concepts provided - for instance for each paradigm presented, a simple virtual machine is also presented (register based, SECD [4] and WAM [7]) and its workings discussed. While this further increases the strain in the lectures, we have found these lectures to be positively received and implementation of the presented virtual machines has become a popular programming project in the class.

The remainder of this paper concerns itself with the largest part of the course, the 7 weeks in which λ calculus and Lisp are used to introduce the functional programming paradigm.

2.1 From λ Calculus to Lisp

True to the idea of treating λ calculus as a programming language, the course opens with the following statement: *You have spent an entire year to learn how to program in a single programming language. In the next 15 minutes, we will teach you another language in its entirety. And in the better part of this course, we will convince you that this language is as powerful as what you already know.*

Definition 2.1. λ expression is:

- variable, a single letter in English alphabet.
- if M is a λ expression and x is variable, then $(\lambda x.M)$ is λ expression
- if M, N are λ expressions, then (MN) is λ expression

The students are then presented with very simple λ calculus definition (Definition 2.1) and the β and α reductions are explained. After we make sure the students understand the basic principles of λ calculus, i.e. they can reduce arbitrary expressions, the course embarks on convincing the students

[1] The english variant of the course with lecture notes, course materials, etc. can be found here: https://courses.fit.cvut.cz/BIE-PPA/index.html

that λ calculus is Turing Complete, or as we state informally[2] *as powerful as any other computer language.*

To this end, students are guided to come up with "challenges" for λ calculus that would prove its expressive power equals that of C or C++, which we do by first removing unnecessary features from C and on the other hand by introducing syntactic sugar and explaining the semantic meaning of the λ calculus. This simultaneous lowering of C and C++ features (objects, redundant forms of loops first, then loops altogether replaced with recursion, etc.) combined with extensions of λ calculus makes it much easier for students to appreciate the abstractions. We have found these exercises very engaging for the students, who in turn learn to appreciate the minimalism in programming language design and the power of purely syntactic extensions.

In this manner first booleans, then Church numerals [2] are introduced as a means to express values and conditional execution and finally the *Y* combinator is introduced together with replacing loops by recursion in the more familiar setting of imperative languages, bridging the gap between λ calculus and the languages students already know. Great care is taken that every newly introduced feature from the simplest (such as λ with multiple arguments) to the most complex ones is explained as a mere syntactic sugar over the basic λ calculus. Only when the students understand the inner workings of these extensions we allow their use. Throughout the course of this transformation, students engage in practical λ calculus "programming" exercises during the tutorials and are encouraged to explore the behavior of more complex λ expressions in their self-study.

When λ calculus is informally shown to be as powerful as programming languages the students already use, the course enters its next phase in which the λ calculus is extended towards ease of programming. Students have so far been carefully guided in the exercises to use the right abstractions to aid in this design so they can actively participate in the next steps based on their own experience. λ calculus is extended with a handful of syntactic shortcuts towards a primitive Lisp-like language. The whole process is highly interactive so that when Lisp is introduced, it is not perceived as a *creative way of nesting parentheses* [3], but as a language, the students helped to "discover" so that they can (and often do) take pride in using it. Finally, with Lisp at our disposal, more features are discussed (*let* expressions, first order functions, Lisp-style macros, etc.), always keeping in mind the connection to the most primitive λ calculus.

The whole segment then concludes with the explanation of the SECD virtual machine, which the students can implement themselves, thus writing a full-fledged interpreter of a practical programming language for the first time, a *life-changing experience* [4] for some.

λ Calculus Evaluators An important aspect of treating λ calculus as a programming language is to write programs in it. While pen and paper, the more traditional method of reducing λ expressions suffices for tiny examples, our goal has always been to enable students to write and experiment on much larger *programs*. The course has thus always relied on λ calculus evaluators, many of which have been created for similar courses at other universities and are available online[5].

2.2 Course Feedback

Each course at FIT CTU is regularly evaluated in a faculty-wide survey after each term and PPA was no exception. The survey evaluates both courses and teachers and contains both numeric and verbal questions. Summary of the three years of PPA is presented in Table 1.

It shows that while the complexity of the course remained the same over the years, the number of students who passed has slightly increased, as did the quality of study materials. Sadly, the overall mark is distorted by inconsistency in the survey over the years, but when compared to other courses, PPA's rank remained roughly the same.

More important, but much harder to visualize were the more complex questions asking students to identify positive and negative aspects of the course. The introductory year was marked by criticism of the study materials. We believe this was due to the fact that the narrative presented in this paper was not firmly established in the first year thoughout the materials. However, the course was praised for its early introduction of functional programming. This changed in the second year of the course, where the strengths remained the same, but criticism shifted from materials to tools, notably the λ calculus evaluators used at the time. As a response, we have introduced LambdaLab [6] as the new λ evaluator, but next year the reported strengths and weaknesses remained roughly the same.

2.3 PPA Specific Survey

The unchanged criticism of λ calculus evaluators prompted us to create a detailed survey in which we asked past students specifically about the role of λ calculus evaluators and their self-study assessment. The turnout was much smaller, partly due to the fact that students have already moved on to different courses, but the responses of 34 students offered us more insight:

[2]Students have not yet been taught about Turing Machines at that point
[3]One of student's comments after the first year survey

[4]another student's informal comment
[5]Lambda Calculator: http://www.cburch.com/lambda/
λ Calculus Interpreter: https://jacksongl.github.io/files/demo/lambda
Lambda-Calculus Evaluator: https://www.cl.cam.ac.uk/~rmk35/lambda_calculus/lambda_calculus.html

Table 1. Summary of the faculty-wide course survey over the past three years. Numbers are averages from 1(best) to 5 (worst). (*) the survey question changed wording from *How useful was the course* to *How much did you like the course*

Term	Students	Replied	Passed	Overall Mark	Contents
Winter 2018	151	48 (31%)	122 (88%)	1.94 (*)	1.92
Winter 2017	192	71 (36%)	163 (84%)	1.56	2.15
Winter 2016	97	37 (38%)	78 (80%)	1.53	2.00

Table 2. Reasons why students did not use provided λ calculus evaluators during the course. The question was a multiple-choice one so the percentages do not add up to 100%

Reason	% of students
different syntax	48%
not aware of them	26%
too complicated	20%
laziness	20%
solved examples more useful	7%
made own evaluator	7%

General Data Our respondents were biased towards students with better marks with students with either A or B overall marks accounting for 53%. This was also reflected in the average points from the λ calculus and Lisp test with an average of 21 and median of 23 (out of 25, the minimum required to pass is 12 points).

Self Study On average, students reported studying λ calculus for 5.25 hours (median of 4), with the longest time reported being 20 hours over the entire term. On average 56% of this time was reported to be spent in exercises (median of 56) as opposed to studying theory. The length of self-study correlated positively (0.15) with the marks.

λ Calculus 70% of the respondents thought that λ calculus was important, while the remaining 30% did not believe that learning it was helpful for their careers. When asked about the main usefulness of studying λ calculus, those who did not find it important mentioned the general knowledge of the term as the main reason, while the other group mostly mentioned its usefulness in understanding the functional programming paradigm and Lisp fundamentals. Only 5% of the students suggested that not teaching λ calculus, or teaching it less would be an improvement for the class[6].

λ Evaluators Only 33% of the respondents actually used the λ calculus evaluators promoted in the class. We asked about reasons for not using the evaluators and present the summary in Table 2. Almost 50% of students indicated that their main reason for not using the evaluators promoted by the course was differences in syntax. Interestingly 26% of the students were not aware of the existence of the evaluators to begin with, which suggests that the evaluators must be integrated into the course materials more tightly for the students to take notice.

Contrary to our expectations, the survey showed that the λ calculus evaluators were neither used nor integrated in the course materials well enough to make a difference. In fact, their effect might have been even detrimental. With the benefit of hindsight we have received hints to the syntax incompatibility issues even before the survey. After learning that many students found the evaluators confusing and their syntax too different from the λ calculus used in the course, we have realized that some of the mistakes we have observed the students made in tests and exams were very close match to what the evaluators would do if the λ expressions were simply copied neglecting the differences[7].

The syntax differences were often subtle, such as different parenthesis requirements (such as required parentheses around λ bodies, or applications), the support for multiple arguments in λ, or lack thereof and so on. The important discrepancy from our point of view was missing support for single letter variable names without any separation, so that $(\lambda xy.\ yx)$ is λ function with two arguments x and y and not a function with single argument xy. We have found that this notion greatly simplifies whiteboard exercises as it makes whitespace redundant. Although we were aware of these issues and warned our students, the survey together with the observed exam errors showed that students were puzzled by the evaluators often deriving completely wrong understanding of λ calculus reductions.

To fix these issues, we have decided to implement our own λ calculus evaluator, in which the problems we had so far can be fixed. We present this work in the next section.

3 Lambdulus

The surveys told us that certain features of an evaluator are a must if it is to be used by the students at all, namely:

- 100% compatible syntax
- clean and intuitive user interface
- zero cost installation
- seamless integration with other course materials
- powerful visualization to demonstrate the effects of reductions

[6] While these numbers look positive, given the bias towards better students in the answers, we hope to be able to achieve better numbers next year

[7] Since internet access is not possible during the tests, we believe the students were learning from the evaluators, but interpreted the results incorrectly

On top of that we believed that the biggest weakness of existing evaluators is their poor scalability towards larger and more complex λ expressions. Tools like LambdaLab [6] are very good in visualizing the expression trees, and displaying the step-by-step reductions of an expression. This works well for simple expressions where the main learning outcome is understanding of where & how to apply given reduction in an expression, but contrasts with our larger goal of treating λ calculus as a programming language, and therefore encouraging students to experiment on much more complex expressions.

Before deciding to write yet another λ calculus evaluator, we evaluated the possibility to simply update existing tools to our needs. Although some of our requirements, such as syntax compatibility would be relatively easy to adapt (although none of the tools we surveyed supported such changes explicitly), we felt that the more complex features, such as visualization, support for large expressions and course integration, which would need to be done from scratch anyways will constitute the bulk of the required work. In the end the matter was decided when a former student of the course[8] who was motivated to extend the simple λ calculus evaluator submitted as course project to a full featured tool.

The next subsections describe how we approached these requirements.

3.1 Architecture

Lambdulus[9] is written in TypeScript. TypeScript was chosen over JavaScript because we wanted the code of the Lambdulus itself to be educational in demonstrating how a λ calculus evaluator can be written. Since our students come predominantly from statically typed C++-like languages, the types and class-based inheritance in Typescript gives the students feeling of familiarity they would not get from dynamically typed prototype based JavaScript.

When designing Lambdulus, we were aware that having a good and *motivated* student at our disposal is nothing short of luck and we wanted to make the decission whether to adapt existing or create own evaluator we faced easier for less fortunate educators. To this end, Lambdulus was, from the very beginning designed with easy extensibility in mind. It has a modular architecture, where relatively simple core deals with λ expression parsing, identification and execution of the reduction steps. We believe that adapting our tool to other courses should only involve limited patches to this simple core module. A web-based front-end implemented as a static webpage written using the React.JS[10] framework then visualizes the reductions performed by the core and deals with user input. Static webpage model was used for its simplicity - hosting is effortless and it is entirely possible for the students to use Lambdulus even when offline from local files or cache.

Dialect As one of the major reasons for not using the existing λ calculus evaluators was differences in used syntax, we have paid very close attention to mimic the syntax of the λ calculus presented in the course in every detail. The basic λ calculus Lambdulus accepts is very simple:

```
EXPR := ident
     |= EXPR EXPR
     |= '(' lambda IDENT { IDENT } '.' EXPR ')'
     |= '(' EXPR ')'
     |= number
     |= operator
```

First three lines directly map to the λ calculus definition and describe an identifier, application and a lambda (with potentially multiple arguments) with body separated by a dot as valid λ expressions. The next rule allows for inserting parentheses around lambda expressions. The last two lines add support for numbers and operators, which do not normally belong to the λ calculus definition. Note that instead of adding new functionality, they simply allow built-in macros (see paragraph below) to be named accordingly. λ with multiple arguments are lowered to single argument counterparts for the core.

Lambdulus supports two different kinds of identifiers. The single letter identifiers mentioned earlier can be followed by a single digit, i.e. $a0a1bc4$ which are used extensively in the course and more traditional whitespace delimited variable names optionally followed by arbitrary number of digits, such as $foo1bar12$. After discussion with TAs of the course we also added the closing bracket] as a special token which closes all currently opened parentheses, i.e. $(x(y(z]a$ is equivalent to $(x(y(z)))a$

3.2 Evaluation

Given a λ expression, the core identifies the appropriate redexes and reduction type and returns the reduction instance containing this information. The reduction can be used by the frontend both to visualize the step properties (such as λ and argument selection) as well as to compute new λ expression corresponding to the reduction performed on the original one. The next paragraphs deal with the more interesting aspects of this evaluation process.

Macros When teaching λ calculus in PPA we taught students to substitute untouched parts of the λ expressions being reduced with capitalized names and only to replace them with their contents when needed for the next reduction steps. To this end, Lambdulus supports a similar notion via *macros*. This concept has been formally introduced by [6] and our basic implementation does not differ. Macros in Lambdulus are either built-in, or user defined. The built-in macros cannot be changed by the user and cover the most

[8] Author of this paper
[9] Available from https://github.com/lambdulus
[10] https://reactjs.org/

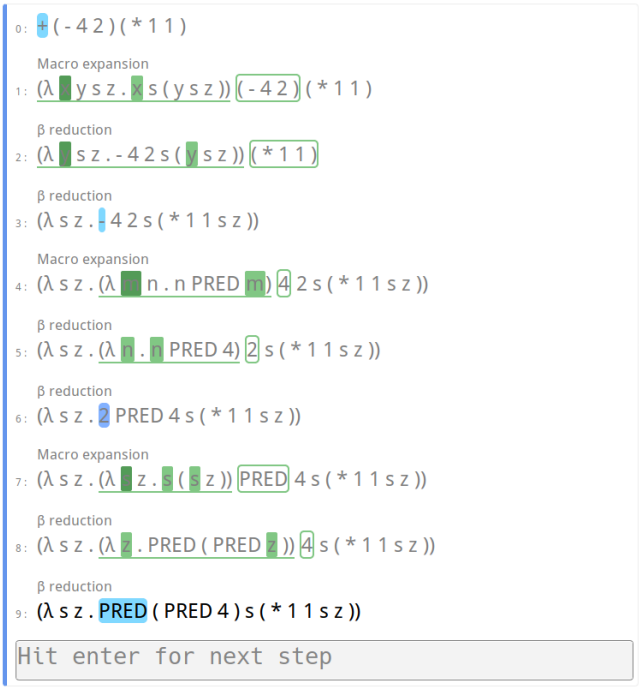

Figure 1. Example of evaluation with lazy expansion of macros. Aarithmetic operators and Church numerals are expanded right before being applied to its arguments (steps 0 and 3, expanded macro is highlighted). Note that macro can expand to other macros, such as the expansion of – into λ expression containing macro *PRED* in step 4. Such macro composition allows students to easily understand the underlying calculations.

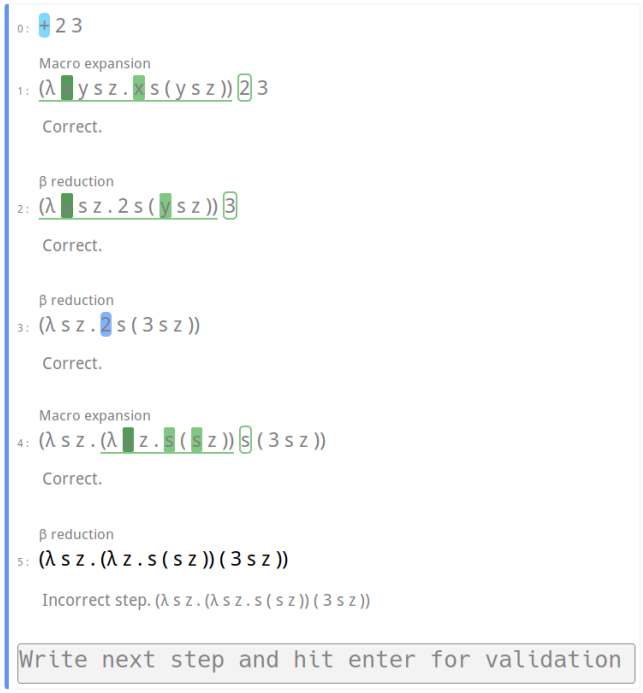

Figure 2. Example of interactive exercise process. User types each step which is then checked for errors. Each step is evaluated for correctness and incorrect steps (such as step 5) are replaced with correct values so that the student is allowed to continue the exercise.

fundamental operations, such as basic checks (*ZERO*, *IF*) and artithmetic operations (+, −, ∗, /) which, if accidentaly changed by students might lead to confusing and hard to debug errors. User defined macros can be created, updated and deleted at any time. Macros are expanded when needed (i.e. when the macro must be substituted for its contents to correctly decide the next reduction step) and their expansion is a separate step, as shown in figure 1.

Dynamic Macros While macros are very useful in simplifying λ expressions, their static nature is a limitation. Consider for instance Church Numerals. While every single number used in an expression can be specified as its own macro, this process does not scale well, nor does it work for recognizing newly created numerals. To fix this, we introduce *dynamic macros*. Their expansion is ruled by the same semantics as normal macros, but the actual expression they substitute is calculated lazily for each expansion and may depend on non λ calculus attributes (such as the numeric value of the desired number). Currently only Church Numerals (built-in macros) are implemented as dynamic macros, but we are investigating ways how to allow user defined dynamic macros as well.

Reductions and Evaluation Orders Lambdulus implements and correctly detects α conversion, β reduction, and η conversion. Choosing between *normal* and *applicative* evaluation order as they are defined in the course lectures is also supported [11].

3.3 Exercise Mode

In order to be able to program in λ calculus, one must first understand the basics. To this end, Lambdulus provides highlighting of current λ, substituted variable and its bound occurrences as well as the argument itself. After each step, the type of performed action is displayed as well. Finally, Lambdulus supports an *exercise mode*, in which no hints are displayed and the student must correctly input the reduced λ expression after each step. Lambdulus then validates the input and displays the highlights as in normal mode. Figure 2 shows an example of an exercised expression.

[11] We are aware of the imprecise definition of evaluation orders wrt λ calculus. It is our goal to use Lambdulus and its visualizations to introduce students to proper definition via reduction strategies in its next iterations, such as [5]

3.4 Debugging Mode

However, the focus of Lambdulus lies not in teaching the mechanics of simple reductions, but to help students to experiment with much larger λ calculus examples. At such complexity, step-by-step evaluation is too tedious. Reduction to the normal state, while fast, does not offer enough information for debugging issues, or if all steps were retained, searching for the relevant part of the evaluation will be too tedious.

To this end, Lambdulus offers *breakpoints*. Similarly to how breakpoints work for programming languages, when a breakpoint is set on a λ expression part when that part of the expression becomes involved in the evaluation process, the evaluation stops. Table 3 summarizes parts of the λ expression on which breakpoints can be set and their effect.

Throughout the evaluation, breakpoints propagate as do the λ expressions they are attached to. Consider $(\lambda x.xx)Z$ with a breakpoint set on Z. After the β reduction, the expression will become ZZ and both Z will have breakpoints set on them. The combination of breakpoints and highlighting the next steps in expressions allows students to understand the behavior of more complex recursive functions, such as *fact(3)* seen in figure 3 and beyond [12].

3.5 Integration with Course Materials

While λ calculus evaluators were only mentioned in the course materials with provided links, Lambdulus goes further. This turned out to be easier than anticipated because we could use the fact that Lambdulus understands the λ calculus dialect used in course. We have therefore changed each provided exercise into a link which opens said expression directly in Lambdulus exercise mode. On top of the tighter integration, we also obtained solutions to the exercises for free as well as caught a few bugs in the exercise notes. We are also adding links to more complex examples to be opened in the debugging mode so that the students can play and explore them interactively. This lead us to the realization that having together documentation *and* the exercises to play with is very helpful and saves the students from constant switching between course materials and Lambdulus. We have thus added the ability to edit and render Markdown to Lambdulus as well.

Continuation of this effort is what we call Lambdulus notebooks, which builds on the idea of literate programming [3], where documentation in markdown form can be combined with various λ expressions, exercises and interactive explorations as shown in figure 4.

To assist students with debugging their issues and sharing their ideas, the active expression is automatically serialized into the URL of the page, therefore simply sharing the current link shares the current state. These URLs can for instance be posted in discussion boards as error reproducibles improving the support experience for the students.

[12] Lambdulus is useful for even more complex examples, such as quicksort on lists, but displaying these via screenshots would serves no purpose

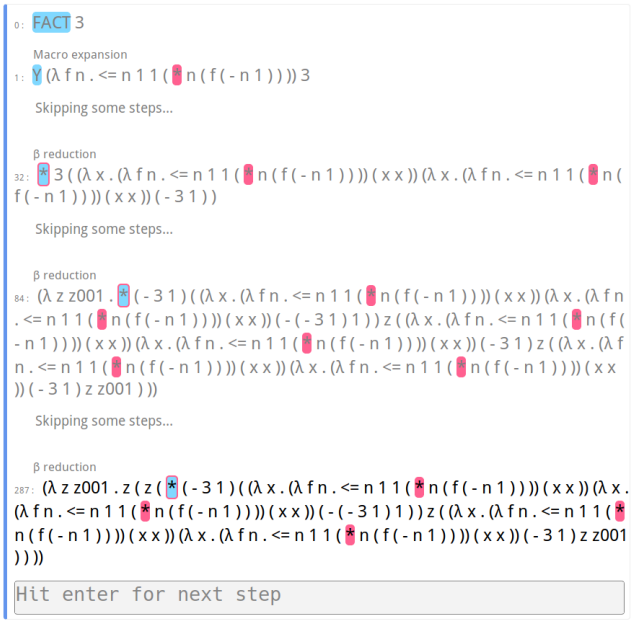

Figure 3. Example of evaluation of *FACT*3 expression with breakpoints. In the first step breakpoint was set on the highlighted multiplication. The next line (step 32) is when the breakpoint was hit. Other breakpoints in the step were automaticaly set by Lambdulus as the breakpoint tags propagate with their objects. Each of these "copies" is a breakpoint of its own and can be hit independently.

4 Future Directions

At the time of submitting this paper, Lambdulus is actively developed to be ready for its introduction in Fall 2019 term. This section first briefly presents the improvements that will be ready by then and then follows with a discussion of our longer-term plan.

4.1 Developed Features

Currently developed features for Lambdulus fall into three broad categories:

Exercise Mode Improvements The exercise mode will support different hints to the students (revealing the reduction of the next step, revealing the actual redex, etc.). We are also working on an expert system able to detect common mistakes (missing α renaming, wrong β redex selection, etc.) to help students learn from their mistakes.

Debugging Mode Improvements The debugging mode is being extended with macro compaction and ad-hoc macros where students select uninteresting parts of the expression and these will be hidden behind a macro valid only for the

Table 3. Parts of λ expressions on which break points can be set and actions before which the breakpoints are triggered.

Target	Stopping Point
any part of λ expression	when substituted in β reduction
argument variable of λ	when renamed in α conversion, or substituted in β
whole λ	when any of its argument substituted or renamed, or when itself substituted as argument to other λ
macro	when expanded, or substitued in β

Figure 4. Lambdulus also supports notebook functionality where exploratory evaluations and exercises can be interleaved with study materials in Markdown as an interactive learning tool.

current expression. Breakpoints on different λ expression parts are extended with configurable triggers (i.e. trigger only on β reduction, not on α renaming, etc.) and with conditions. Finally, we are extending the evaluator to retain information about substituted λ functions and values substituted for their arguments so that these can be displayed when execution is stopped to help students orient in even more complex examples.

Course Integration Inspired by [8] we are extending Lambdulus with tracing, collecting metrics about the individual students' progress, time spent in different modes and with different exercises, errors made, and so on to offer the educators a real-time preview of the class status and to provide rankings to motivate the students further.

4.2 Long Term Goals

In the longer run, we want to extend Lambdulus along the following two main directions:

Formal λ Calculus Because we introduce λ calculus to very young students, we hide many of the interesting details and use simpler, yet less precise notions, such as evaluation strategies, etc. We hope that with proper tools, we can bridge the gap and explain even those harder concepts. Furthermore, this would make Lambdulus interesting not just for introductory, but also for more rigorous advanced courses.

Lisp As of now, Lambdulus supports the λ calculus dialect as it is described in the first part of the course. However, as described in section 2.1, we further extend λ calculus with additional features, to make it more Lisp-like all the way to proper Lisp. We would like to mimic these in Lambdulus itself. Namely, we would like to make the Lambdulus syntax whitespace aware so that the expressions can be structured for better readability, improve its parser and editor to offer helpful error messages and hints, and allow for the inline definition of recursive macros. Finally, we would like to extend Lambdulus with a Lisp core allowing students to explore another language with its features in a familiar environment. Although Lisp can indeed be simulated in λ calculus entirely, we feel that following the lectures and switching to the different core will be more natural. Specifically, we would like to add Lisp macros where visualization of the program-as-data and compile vs runtime execution in Lambdulus can have the greatest impact. The Lisp addition will be finished by implementing an SECD virtual machine visualization in Lambdulus as well.

5 Conclusion

Teaching λ calculus is often left for advanced programming language courses, partly due to its theoretical nature. We have challenged this perception and introduced λ calculus to second-year undergraduate students. To do so successfully, we changed the way λ calculus is presented and treated it not as a theoretical concept, but as a real and practical language. This required both changes to the way λ calculus is taught as well as the development of tools that allowed to really treat it as proper programming lanuage. We thus presented the outline of our course and discussed how λ calculus is approached as a programming language from the very beginning. We also introduced Lambdulus, λ calculus evaluator with emphasis on supporting and visualizing evaluation of much larger and more complex λ expressions.

We believe that the idea of approaching theoretical concepts practically first, letting students "discover" the theoretical implications by themselves via appropriate tooling and guidance has its merits and can be applied to much wider range of concepts. We hope that our tools can benefit other similar classes as much as we ourselves, and transitively our students will benefit from discussions with other educators our work might inspire.

Acknowledgments

Supported by the Czech Ministry of Education, Youth and Sports from the Czech Operational Programme Research, Development, and Education, under grant agreement No. *CZ*.02.1.01/0.0/0.0/15_003/0000421. We are also grateful to the reviewers for their constructive input. This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 695412).

References

[1] Alonzo Church. 1936. A note on the Entscheidungsproblem. *The journal of symbolic logic* (1936).

[2] Alonzo Church. 1941. The Calculi of Lambda-Conversion. *Annals of Mathematics Studies* 6 (1941).

[3] D. E. Knuth. 1984. Literate Programming. *Comput. J.* 27, 2 (01 1984), 97–111. https://doi.org/10.1093/comjnl/27.2.97 arXiv:http://oup.prod.sis.lan/comjnl/article-pdf/27/2/97/981657/270097.pdf

[4] P. J. Landin. 1964. The Mechanical Evaluation of Expressions. *Comput. J.* 6, 4 (01 1964), 308–320. https://doi.org/10.1093/comjnl/6.4.308 arXiv:http://oup.prod.sis.lan/comjnl/article-pdf/6/4/308/1067901/6-4-308.pdf

[5] Gordon Plotkin. 1975. Call-by-Name, Call-by-Value and the Lambda Calculus. In *Theoretical Computer Science*, Vol. 1. 125,159.

[6] Daniel Sainati and Adrian Sampson. 2018. LambdaLab: An Interactive λ-Calculus Reducer for Learning. In *ACM SIGPLAN SPLASH-E Workshop (SPLASH-E 18)*. https://doi.org/10.1145/3310089.3313180

[7] David H. D. Warren. 1983. *An Abstract Prolog Instruction Set.* Technical Report. Artificial Intelligence Center, Computer Science and Technology Division, SR International. http://www.ai.sri.com/pubs/files/641.pdf

[8] T. Wrigstad and E. Castegren. 2017. Mastery Learning-Like Teaching with Achievements.. In *ACM SIGPLAN SPLASH-E Workshop (SPLASH-E 17)*. http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-333724