Haxe as a Swiss knife for bioinformatic applications: the SeqPHASE case story

Yann Spöri
yann.spori@ulb.be

Jean-François Flot
jean-francois.flot@ulb.be

October 15, 2023

1 Abstract

Haxe is a general purpose, object-oriented programming language supporting syntactic macros. The Haxe compiler is well known for its rather unique ability to convert the source code of Haxe programs into the source code of a variety of other programming languages including Java, C++, JavaScript and Python. Although Haxe is becoming more and more used for a variety of purposes, including games, it has not yet attracted much attention from bioinformaticians. This is surprising, as Haxe allows generating different versions of the same program (e.g. a graphical user interface version in JavaScript running in a web browser for beginners and a command-line version in C++ or python for increased performance) while maintaining a single code, a feature that should be of interest for many bioinformatic applications. To demonstrate the usefulness of Haxe in bioinformatics, we present here the case story of the the program SeqPHASE, written originally in Perl (with a CGI version running on a server) and published in 2010. As Perl+CGI is not desirable anymore for security purposes, we decided to rewrite the SeqPHASE program in Haxe and to host it in Github Pages (`https://eeg-ebe.github.io/SeqPHASE/`), thereby alleviating the need to configure and maintain a dedicated server. Using SeqPHASE as an example, we discuss the advantages and disadvantages of Haxe's source code conversion functionality when it comes to implementing bioinformatic software.

2 Introduction

Few biologists are proficient in using command-line tools [1]. As a result, bioinformatic software needs to be usable without the need to open a terminal. Nevertheless, some computer-savvy users prefer interacting with a tool via commandline interfaces [2] (CLI) or they do need a command-line version in order to integrate the tool into a pipeline such as Galaxy [3–5]. Thus in practice most bioinformatic programs require two interfaces - a graphical user interface (GUI) and a CLI.

Even though it is pretty straightforward to program a CLI, adding a GUI to a program can be trickier. GUIs can either be provided in form of a standalone application or via an external program such as a web browser (e.g. Chrome, Firefox, Edge or Safari). Although specialised toolkits (like Swing, SWT or FX for Java, Flutter for Dart/C++) exists that allow the creation of platform-independent standalone GUIs, this solution requires installing and maintaining the corresponding piece of software. By contrast, a web browser is preinstalled on most operation systems and is therefore more practical for biologists to use.

Historically, embedding code into a website was usually done using Java Applets or Flash applications with the Netscape Plugin Application Programming Interface [6]. However, newer web browsers do not allow this integration anymore due to security concerns [7]. Instead, modern browsers only allow the interpretation of a particular set of programming languages, namely webassembly [8] and ECMAScript (with its better known dialect JavaScript) [9].

Due to this limitation most programming languages cannot directly execute code inside a web browser. Thus, when programmers want to avoid maintaining two distinct versions of the same software (one written in e.g. JavaScript and the other one in Python), there are three possible ways to write a program with both a command-line interface and a graphical user interface running in a web browser:

- writing the whole program in JavaScript, then using e.g. NodeJS (`https://nodejs.org/`) to execute the JavaScript program in a terminal environment. However, high-level programming constructs such as classes are rather awkward to use in JavaScript, except with the help of a scripting language such as CoffeeScript [10];

- writing a program with a CLI that runs on a web server. The graphical user interface can then communicate with the program running on the web server and visualize its results (e.g. via BioJS [11]). Many scripting language such as PHP, Perl, Python or Ruby support communication via the common gateway interface (CGI)[12]. Nevertheless, setting up and maintaining such a dedicated public server can be time and resource-costly. It also requires users' data to be sent over to the server via internet, which can be a problem in case of large and/or sensitive datasets;

- using a programming language that supports not only the interpretation of converting the source code into a binary program but also allows the conversion of the source code into the source code of various other programming languages (a process variously called "trans-compiling", "transpiling" or "cross-compiling" depending on authors). A JavaScript version of the code can then run inside a web browser while another version of the program (e.g. in Python) can be used in a terminal environment.

Haxe is a general purpose, highly versatile object-oriented programming language [13]. The source code of Haxe programs can be converted into the source code of a variety of other languages including Java, C++, JavaScript and Python [14]. Using Haxe, it is thus relatively easy to create multiple versions of the same tool - e.g. a JavaScript version that runs inside a web browser and a Python version that can run as a terminal application. Furthermore, Haxe supports sophisticated programming paradigms such as syntactic macros, making it an excellent choice for the third path described in above paragraph.
To illustrate the usefulness of Haxe as a Swiss knife for bioinformatic applications, we tell here the story of how we used Haxe to revive an old but very useful piece of code, SeqPHASE [15], originally written in Perl + CGI code, by reimplementing it in Haxe then converting the Haxe code into JavaScript (for the graphical version) and python (for the command-line one).

3 Material and methods

SeqPHASE was written in 2010 to address a pressing problem in population genetics: using PHASE [16] (a program originally written to infer the most probable pairs of haplotypes in a population of diploid organisms for which genotypes have been determined, but designed for microsatellites and not DNA sequences) on FASTA files, and converting the output of PHASE back into FASTA [15].
Until that point, a tool allowing this conversion existed as part of the Windows program DNAsp [17, 18], but with several severe shortcomings [15], and besides conversion was impossible on other operating systems such as Linux or Mac.
Because it filled an important need in the biological community, SeqPHASE was immediately adopted and cited a relatively large number of times (about 500 times since published in 2010).
However, its implementation in Perl + CGI posed important security issues, and after a few years university computer infrastructure administrators started to become very reluctant to host this piece of code on public servers that could offer an entrance point to hackers.
Out of necessity, it was therefore decided to reimplement SeqPHASE completely; this time in Haxe as we needed to provide both a user-friendly web browser tool and a command-line version for use in pipelines and on computer clusters.
For the sake of simplicity and cost-saving, choice was made to host the source code on GitHub (`https://github.com/eeg-ebe/SeqPHASE`) as a mix of Haxe (for the core of the program) as well as JavaScript, HTML and CSS (for the user interface).
Cross-compiling the Haxe part of the code into JavaScript resulted in a fully functional web browser tool accessible via GitHub Pages (`https://eeg-ebe.github.io/SeqPHASE/`), whereas cross-compiling the same Haxe code into Python produced a cross-platform Python script (with a CLI) made available for download on the same website (`https://eeg-ebe.github.io/SeqPHASE/download.html`).

4 Results

The source code of the reimplemented SeqPHASE program is available at `https://github.com/eeg-ebe/SeqPHASE` and licensed under the Apache 2.0 license. A web page allowing the user to run and/or download the program is available on GitHub Pages `https://eeg-ebe.github.io/SeqPHASE/`.
The reimplementation consists of the following files:

- a series of static HTML pages including a menu page, a FAQ page and so on;
- two dynamic web pages where users can run respectively for the first (FASTA to PHASE) and second step (PHASE to FASTA) of the SeqPHASE program. The web pages will directly call the corresponding JavaScript code;
- a download page that allows users to download a zipped archive of the GUI version of the program (once unzipped, users can run SeqPHASE offline inside a web browser by double-clicking the index.html file found inside that archive) as well as Python command-line versions of the two steps of SeqPHASE.

During the reimplementation process we discovered small bugs in the original code and corrected them, namely:

- FASTA to PHASE: Every sequence in the inputted FASTA files should have the same length. However it was possible to overcome this check by ordering the sequences by sequence length in descending order.
- PHASE to FASTA: The sorting algorithm now first groups the sequences of individuals together and then sorts these groups. That way the sequences of individuals are not sorted apart if the name of one sequence is a full prefix of a sequence with a longer name (e.g. the name of sample1a and sample1b would be sorted apart if there is another sequence with the name sample1abc1a).

Although it is unlikely that these bugs had an impact on the accuracy of the results, it sometimes resulted in the program running on inputs that contained errors that should have been reported.

5 Time usage

The different programs running on a computer at the same time are competing for the hardware resources present (CPU, network resources, hard disc access, RAM, ...). Since the operation system does not always prioritize the access of all hardware resources to one particular program the exact time a particular program needs may vary. Another problem is that temperature changes can - among other things - negatively affect the computers clock accuracy [19]. This can lead to time measurement errors, especially when it comes to short-running programs.
Although in computer science it is common to use mathematical approaches ($\mathcal{O}$-Notation [20]) to evaluate the running time of different algorithms, this approach is not really usable here, since all programs implement the same algorithm (converted to different programming languages via Haxe). Thus all programs exhibit the same asymptotic runtime behavior.
Thus in order to determine the "relative running time" we let Haxe create a C++, Lua, Neko, Python, JavaScript (NodeJS) and Java version of the program. We then launched these programs as well as the original Perl program 10,000 times respectively and measured the corresponding time usages (compare figure 1 and table 1). Hereby the 140,000 program executions (=10,000 runs * 7 program versions * 2 conversion steps) were executed in an asynchronous manner and in random order in order to spread hardware resource access related problems as well as temperature change issues the most evenly possible over all time measurements.
When comparing the average calculation time needed to convert a FASTA file to a PHASE input file or a PHASE output file back to a FASTA file, the C++ version greatly outperforms all other versions of the SeqPHASE program, followed by the Lua version, the (original) Perl version, the Neko version, the Python version, the NodeJS version and last but not least the Java version. This order is statistically

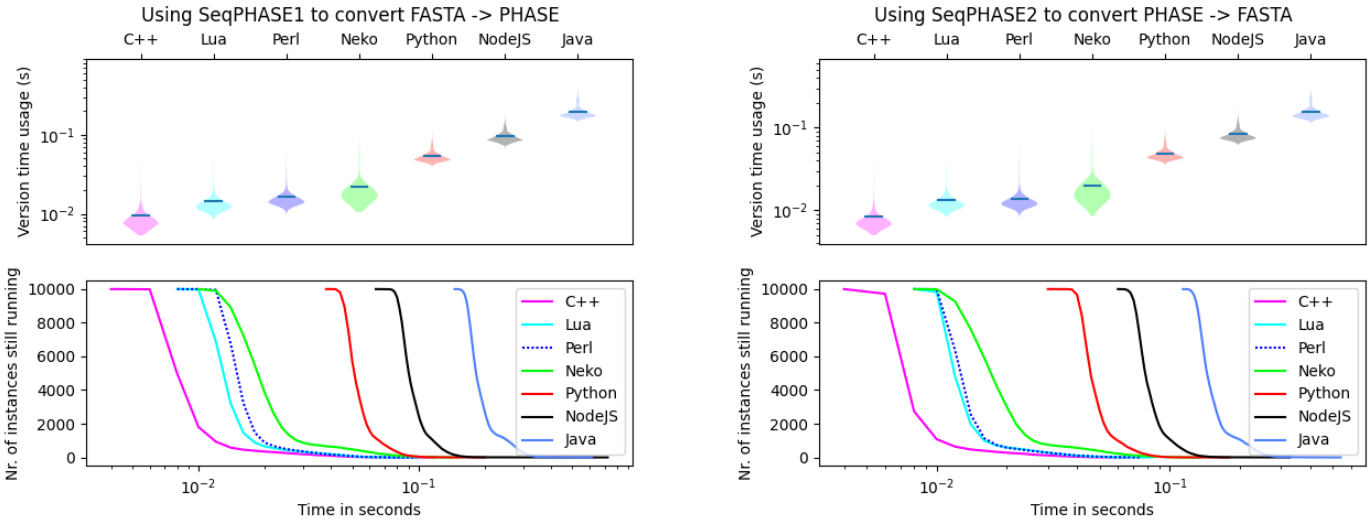

Figure 1: Speed measurements of the different tasks and versions of the SeqPHASE programs. The left subfigure analyses the calculation time needed to convert a FASTA file to a PHASE input file while the right subfigure analyses the calculation time (in seconds) needed to convert a PHASE output file back to a FASTA file. For each of these two subplots we launched the different SeqPHASE program versions 10,000 times. Then we plotted violin plots showing the time usage of each program version as well as a plot showing the number of program instances for the different program versions still running at a particular timestamp after the corresponding program launch. The minimum, maximum, standard deviation, average and median of these runs for the different program versions and conversions are listed in table 1.

Program	FASTA -> PHASE					PHASE -> FASTA				
	Min	Max	σ	Avg	Med	Min	Max	σ	Avg	Med
C++	5.36	116.59	5.58	9.39	7.99	5.0	65.96	4.4	8.3	7.18
Lua	8.67	97.08	6.32	14.48	12.86	8.33	82.22	5.63	13.44	11.89
Perl	9.6	96.53	6.0	16.37	14.82	8.21	74.09	5.47	13.96	12.51
Neko	10.56	148.64	12.88	21.77	18.57	8.59	135.98	11.6	19.91	17.09
Python	39.55	199.11	9.96	54.02	51.03	31.22	179.77	9.29	48.64	45.73
NodeJS	65.93	722.73	17.04	95.25	90.29	60.58	327.83	14.37	82.95	78.5
Java	146.54	612.77	37.25	195.61	182.62	114.31	541.71	32.07	155.45	144.67

Table 1: Time usage (in ms) of the different tasks and versions of the SeqPHASE programs. The table lists the minimum, maximum, standard deviation, average and median time the Perl, C++, Lua, Python, Neko, NodeJS and Java program needed to convert a FASTA file to a PHASE input file (left part of the table) and a PHASE result file back to a FASTA file (right part of the table).

highly significant as Welch's unequal variances t-test yields extremely low p-values for each version pair. In fact, the highest p-value with a value of 3.2E-11 can be observed for the pair Perl & Lua when it comes to the conversion of PHASE output files to FASTA files.

Although the C++ version is the fastest we decided to not provide it as we found it challenging to compile a C++ program that runs on all computers. Instead we opted for the Python version because Python is a well known programming language already installed on most computers and because speed is most often not that much of an issue when it becomes to small, straightforward programs that are running less then 100ms. In case a high performance C++ version is needed nevertheless we suggest building a C++ version for a particular target computer with platform-specific code optimisation.

Even though this assumption still needs to be verified, we believe that Chrome and NodeJS will need, on average, the same amount of time to execute the SeqPHASE program since both programs are using the same JavaScript engine (named V8 [21]).

Despite the fact that Java is usually faster than Python, it performed worse in our comparison. Whenever a Java program is executed, a Java Virtual Machine (JVM) is first launched which is then used to run the Java program. However when converting small and medium sized input files, the process of launching the

JVM is slower then executing the entirety of the Python SeqPHASE program. It is thus likely Java will gain the upper hand for larger input files that take more time to process or on computers that take less time to start the JVM. In this context, it should be mentioned that Neko code also runs inside a virtual machine (VM), but the Neko VM takes significantly less time to start.

6 Discussion

Through the example of SeqPHASE's reimplementation, we demonstrate that Haxe coding is a valuable yet underused approach for bioinformaticians to make their programs available online to large audience made up of a mix of biologists with no command-line proficiency and computer scientists requiring command-line tools to run on computing clusters.
The SeqPHASE website offers a template for users wishing to replicate such a website for other tools they have designed, and offers therefore a valuable alternative to CGI apps, for instance. Since websites are running inside a sandboxed web brower having a tool that runs inside a website can also be a huge advantage for users that do not want to install a particular software locally due to security concerns.
Compare to Perl, Python and other programming languages traditionally used in bioinformatics, Haxe suffers from certain drawbacks: (1) the current unavailability of a "BioHaxe" library of functions facilitating the import and processing of biological data. It is our hope to provide such a library in the future. (2) In order to debug a particular program you may need to debug the target language. This can be challenging in case you do not know that particular programming language. (3) Haxe does not support yet certain target languages that are particular common in bioinformatics (such as R & Perl).

References

[1] Olga G. Troyanskaya. Don't fear the command line! *Cell*, 144(6):842–843, 2011. doi: 10.1016/j.cell.2011.02.042.

[2] Artem Voronkov, Leonardo A Martucci, and Stefan Lindskog. System administrators prefer command line interfaces, don't they? an exploratory study of firewall interfaces. In *Fifteenth Symposium on Usable Privacy and Security (SOUPS 2019)*, pages 259–271, 2019.

[3] Jeremy Goecks, Anton Nekrutenko, James Taylor, and Galaxy Team team@ galaxyproject. org. Galaxy: a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences. *Genome biology*, 11:1–13, 2010.

[4] Daniel Blankenberg, Gregory Von Kuster, Nathaniel Coraor, Guruprasad Ananda, Ross Lazarus, Mary Mangan, Anton Nekrutenko, and James Taylor. Galaxy: a web-based genome analysis tool for experimentalists. *Current protocols in molecular biology*, 89(1):19–10, 2010.

[5] James Taylor, Ian Schenck, Dan Blankenberg, and Anton Nekrutenko. Using galaxy to perform large-scale interactive data analyses. *Current protocols in bioinformatics*, 19(1):10–5, 2007.

[6] Tim Lammarsch, Wolfgang Aigner, Alessio Bertone, Silvia Miksch, Thomas Turic, and Johannes Gärtner. A comparison of programming platforms for interactive visualization in web browser based applications. In *2008 12th International Conference Information Visualisation*, pages 194–199, 2008. doi: 10.1109/IV.2008.34.

[7] Ramakrishna Ayyagari and Norilyz Figueroa. Is seeing believing? Training users on information security: Evidence from Java Applets. *Journal of Information Systems Education*, 28(2):115, 2017.

[8] Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up to speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 185–200, 2017.

[9] Allen Wirfs Brock and Brendan Eich. Javascript: the first 20 years. *Proceedings of the ACM on Programming Languages*, 4(HOPL):1–189, 2020.

[10] Trevor Burnham. Coffeescript: accelerated Javascript development. *CoffeeScript*, pages 1–124, 2015.

[11] John Gómez, Leyla J García, Gustavo A Salazar, Jose Villaveces, Swanand Gore, Alexander García, Maria J Martín, Guillaume Launay, Rafael Alcántara, Noemi Del Toro, et al. BioJS: an open source JavaScript framework for biological data visualization. *Bioinformatics*, 29(8):1103–1104, 2013.

[12] Shishir Gundavaram. *CGI Programming on the World Wide Web*. O'Reilly & Associates, 1996. ISBN 978-1-56592-168-9.

[13] Benjamin Dasnois. *haXe 2 Beginner's Guide*. Packt Publishing Ltd, 2011. ISBN 1-84951-257-4.

[14] Domagoj Štrekelj, Hrvoje Leventić, and Irena Galić. Performance overhead of haxe programming language for cross-platform game development. *International journal of electrical and computer engineering systems*, 6(1):9–13, 2015.

[15] Jean-François Flot. SeqPHASE: a web tool for interconverting PHASE input/output files and FASTA sequence alignments. *Molecular Ecology Resources*, 10(1):162–166, 2010. doi: 10.1111/j.1755-0998.2009.02732.x.

[16] Matthew Stephens, Nicholas J. Smith, and Peter Donnelly. A new statistical method for haplotype reconstruction from population data. *The American Journal of Human Genetics*, 68:978–989, 2001. doi: 10.1086/319501.

[17] Julio Rozas and Ricardo Rozas. DnaSP, DNA sequence polymorphism: an interactive program for estimating population genetics parameters from DNA sequence data. *Computer Applications in the Biosciences*, 11(6):621–625, 1995.

[18] Julio Rozas, Albert Ferrer-Mata, Juan Carlos Sánchez-DelBarrio, Sara Guirao-Rico, Pablo Librado, Sebastián E Ramos-Onsins, and Alejandro Sánchez-Gracia. DnaSP 6: DNA sequence polymorphism analysis of large data sets. *Molecular Biology and Evolution*, 34(12):3299–3302, 2017. doi: 10.1093/molbev/msx248.

[19] Hicham Marouani, Michel R Dagenais, et al. Internal clock drift estimation in computer clusters. *Journal of Computer Networks and Communications*, 2008, 2008.

[20] Kalle Rutanen. O-notation in Algorithm Analysis. *arXiv preprint arXiv:1309.3210*, 2013.

[21] Martin Heller. What is Node.js? the JavaScript runtime explained. *InfoWorld*, 2017.