Chemoinformatics and structural bioinformatics in OCaml
Francois Berenger1*, Kam Y. J. Zhang2 and Yoshihiro Yamanishi1,3

Abstract
Background: OCaml is a functional programming language with strong static types, Hindley–Milner type inference and garbage collection. In this article, we share our experience in prototyping chemoinformatics and structural bioinformatics software in OCaml.

Results: First, we introduce the language, list entry points for chemoinformaticians who would be interested in OCaml and give code examples. Then, we list some scientific open source software written in OCaml. We also present recent open source libraries useful in chemoinformatics. The parallelization of OCaml programs and their performance is also shown. Finally, tools and methods useful when prototyping scientific software in OCaml are given.

Conclusions: In our experience, OCaml is a programming language of choice for method development in chemoinformatics and structural bioinformatics.

Keywords: Chemoinformatics, Structural bioinformatics, Bisector tree, Scientific software, Software prototyping, Open source, Functional programming, OCaml

Introduction
There are several schools of thought in computer programming. Each school is represented by several programming languages and some languages are multi-paradigm.

In declarative languages (like SQL), a programmer writes a kind of mathematical specification of what to compute, and the compiler will automatically derive a program implementing this specification. Prolog [1], is also such a programming language where the specification is given as a collection of logic predicates.

On the contrary, in imperative programming, the programmer writes in extensive details how to compute the result he wants. Ada, C, Fortran and Pascal are famous representatives of this style of programming.

In Object-Oriented programming, data structures and the allowed operations on them are grouped into classes. Classes can be hierarchically organized, and behavior inherited so that generic code can be reused between software components. C++, Java, Eiffel, Ruby and Python are famous members of this family of languages. Most Object-Oriented languages use the imperative style of programming.

In functional programming, a program is a collection of functions. State passing is done explicitly via function parameters. Functional programming has a mathematical taste and dates back to Lisp. Lisp, Scheme, OCaml, F#, Haskell, Scala, Racket and Clojure are representatives of the functional style of programming. There are several advantages to using functional programming [2]. Since state passing is explicit, functional programs are easy to reason about. They easily fit in the head of the programmer. Some functional programming languages are pure (e.g. Haskell); they guarantee referential transparency, the fact that an expression can be replaced by its corresponding value without changing the program behavior. There are some articles about the productivity boost associated with functional programming [3, 4].

While there are not many, some functional programming libraries for chemoinformatics do exist. In Haskell, the ‘smiles’ library [5] provides full support for the OpenSMILES specification [6]. While the ‘radium’ library

*Correspondence: beren314@bio.kyutech.ac.jp
1 Department of Bioscience and Bioinformatics, Faculty of Computer Science and Systems Engineering, Kyushu Institute of Technology, Iizuka, Fukuoka, Japan
Full list of author information is available at the end of the article
[7] provides the periodic table plus readers and writers for SMILES and condensed formulas. In Scala, the ‘chem’ library [8] provides a purely functional cheminformatics toolkit [9].

In this article, we concentrate on Objective Caml (OCaml [10]), in the context of scientific software prototyping for chemoinformatics and structural bioinformatics. OCaml is a general purpose functional programming language developed at INRIA, the French national research institute for computer science, robotics and applied mathematics. OCaml focuses on expressiveness and safety. Some of the language’s strengths include its type system, with parametric polymorphism (called generics in Java, templates in C++) and type inference. Thanks to Hindley–Milner type inference [11, 12], the OCaml programmer is freed from explicitly providing function parameters and result types. For a course on programming languages and types, we refer interested readers to Pierce [13]. OCaml supports user-defined algebraic data types, records, sums/enums and pattern matching. Pattern matching is a generalization of the switch statement present in other languages. When pattern matching, a program is driven by the type of the parameter being matched upon. In OCaml, memory is managed automatically, by an incremental garbage collector, preventing memory corruption. Interactive use of OCaml is possible via a read-eval-print loop called the OCaml interpreter. Interacting with the interpreter is a standard way to test a function or to check that some functionality provided by a library works the way one understands it. In addition to its byte-code compiler and interpreter, OCaml offers a compiler that produces efficient executables. Tail-recursive functions are automatically translated to efficient loops by the OCaml compiler. OCaml also features an object-oriented layer, with multiple inheritance, parametric and virtual classes. While OCaml was initially used to develop symbolic computing applications, such as automatic theorem provers, compilers, interpreters and static program analyzers, it is now used to develop software in many other areas.

Functions are first-class values in OCaml. A function can be passed as an argument to, or returned by, another function. OCaml is a multi-paradigm language. For performance reasons, OCaml offers many imperative features (exceptions, modifiable variables, records, arrays and loop statements). OCaml built-in data types include not only integers, floating point numbers, booleans, characters and strings but also more advanced data types such as tuples, records, arrays and lists.

Large programs are easy to structure due to modules, which share some traits with classes in object-oriented programming. Modules can be organized hierarchically and parameterized over a number of other modules. Such a function, from modules to module is called a functor and allows high level generic programming.

OCaml’s evaluation strategy is strict. All parameters to a function are evaluated prior to entering the function's body. The compilation of OCaml programs is fast. For example, the ∼ 3000 OCaml lines (without comments) of the consent software [14] and its four executables compile from scratch and link in ∼ 3.8 s (resp ∼ 1.1) using dune (version 1.6.2) and a single core (resp. up to all cores) of our desktop computer (16 cores, Intel Xeon 2.1GHz, 64GB RAM, Linux Ubuntu 18.04.1 LTS).

Strong static types are types which are enforced by the compiler. Due to the use of types and garbage collection, several run-time errors which plague other programming languages are absent from OCaml programs: null pointer exception, dereference after free, type cast exception, segmentation fault, unhandled switch cases and most memory leaks. In functional programming, more complex properties can be encoded and statically enforced by structuring code using monads [15], which are pervasive in Haskell [16], or by using dependent types (not available in OCaml, but in Coq [17], Idris [18, 19] and Agda [20]). A function that is guaranteed to produce a result in a finite time is called total. Functions for which there is no such guarantee are called partial. For some functions, Idris can check if they are total. However, such advanced functional programming concepts are out of the scope of this article.

Despite not being very popular, OCaml is not a niche language. Most of its academic users work in computer science, on compilers and formal methods. But, OCaml is also used in bioinformatics [21–23], structural bioinformatics [24–27], chemoinformatics [14, 28], systems biology [29–32] and ecotoxicology [33].

There are several industrial users of the language [34] including Bloomberg, Citrix, Dassault Systèmes, Facebook [35], Jane Street (a proprietary high frequency trading firm) and Microsoft.

OCaml has some successes in the industrial world: Lexifi’s Modeling Language for Finance [36], the ASTRÉE Static Analyzer [37] used by Airbus to certify on-board software and Microsoft’s static driver verifier [38]. OCaml has several successes in the open-source world too: the Unison file synchronizer [39], the MLdonkey [40] multi-protocol peer-to-peer client, the Coq [17] proof assistant [41] and FFTW’s symbolic optimizer of fast Fourier transforms [42].

In the remaining of this article, resources to learn OCaml are listed in “Resources to learn OCaml” section. Explanations on types and how to read signatures of OCaml functions are given in “Understanding OCaml type signatures” section. Tools for proficiency in OCaml are listed in “An OCaml programming environment”
section. Several uses cases of OCaml in Chemoinformatics and Structural Bioinformatics are given in “OCaml in chemoinformatics and structural bioinformatics” section. The parallelization of scientific programs is dealt with in “Accelerating chemoinformatics and structural bioinformatics in OCaml” section. Finally, strengths and weaknesses of the language and ecosystem are discussed (“Scientific software prototyping in OCaml” and “OCaml language and ecosystem drawbacks” sections), before concluding.

Methods

Resources to learn OCaml

There are several books introducing the language [43–45], some of them freely available online [46–48]. Other books [49, 50] give an excellent introduction to functional programming.

The “Caml Trading” video, a talk given at Carnegie Mellon university [51], explains in details why OCaml was chosen by a high frequency trading firm [52, 53]. Like researchers, this company has the technical requirements of correctness, agility and performance.

To give a try at the language within a browser, OCaml-PRO offers an OCaml interpreter and some basic lessons [54]. To learn the language via the official documentation online [55], here are the essential chapters: Chapter 1 “The core language”, Chapter 2 “The module system”, Chapter 4 “Labels and variants”, The Pervasives module (a set of functions which is always available to the programmer), The list module (the most useful data structure in functional programming). One should be able to start programming in OCaml after having read only this material.

The standard library documentation is available online [56]. While it allows one to have an idea of the standard modules and their capabilities, it is not recommended for large scale software development. For real world programming, an extended standard library is necessary. For example OCaml-containers (code [57] and documentation [58]) or OCaml batteries-included (code [59] and documentation [60]) or Janestreet’s core (code [61] and documentation [62]).

To give a taste of OCaml, Fig. 1 shows the complete definition of a bisector-tree [63]. A bisector tree is a data structure to store n-dimensional points provided a distance function between those points exists. Such a tree allows to do fast nearest neighbor searches and orthogonal queries [64]. Vantage point trees [65, 66] and μ-trees [67] are closely-related data-structures which could be used for the same purpose. Our implementation (opam package bst [68]) is parameterized by a distance function and bucketized, i.e. leaves of a tree can hold up to $k \geq 1$ (user-chosen parameter) molecules.

Understanding OCaml type signatures

A type signature is a formal specification of the behavior of a function. Unfortunately, most of the time, this specification is incomplete and unless the function’s name is explicit enough, reading the documentation is necessary to understand the complete specification.

Since being able to read type signatures is essential in OCaml, we list in code as well as in plain English some of the type signatures of essential functions of the list module. The list module uses polymorphic types, i.e. a list can contain elements of any type, but a given list can only contain elements of the same type. $\alpha$ and $\beta$ are standard names for polymorphic types.

For brevity later on, a few definitions are given hereafter.

Definition 1

The syntax

\[
apply : \alpha \rightarrow \beta
\]

defines the type of a function named apply from type $\alpha$ to type $\beta$ in which $\alpha$ and $\beta$ are type parameters. The equivalent C++ header file portion would be

\[
\text{template } \langle \text{class A, class B}\rangle \\
B \text{ apply (A a)};
\]

Definition 2

Let’s call accumulate any function which takes an $\alpha$, a $\beta$ and returns an $\alpha$.

\[
\text{accumulate} : \alpha \rightarrow \beta \rightarrow \alpha
\]

Definition 3

Let’s call side-effect any function which takes an $\alpha$ and returns nothing (in OCaml, nothing’s type is called unit).

\[
\text{side-effect} : \alpha \rightarrow \text{unit}
\]
Definition 4  Let’s call predicate any function which takes an \( \alpha \) and returns a Boolean.

\[
\text{predicate}: \alpha \rightarrow \text{bool}
\]

Definition 5  Let’s call comparison any function which takes two alphas and returns an integer.

\[
\text{comparison}: \alpha \rightarrow \alpha \rightarrow \text{int}
\]

Then, it becomes possible to explain some list functions and their type signatures.

\[
\text{cons}: \alpha \rightarrow \alpha \text{list} \rightarrow \alpha \text{list}
\]

The cons (construct) function takes an \( \alpha \), a list of alphas and returns a list of alphas. The \( :: \) syntax operator is also available for the cons function. Hence, the OCaml expression \(1 :: [2;3;4]\) constructs the list \([1;2;3;4]\) and \(\alpha = \text{int}\).

\[
\text{hd}: \alpha \text{list} \rightarrow \alpha \text{list}
\]

hd (head) takes a list of alphas and returns an \( \alpha \) (the first one in the list).

\[
\text{tl}: \alpha \text{list} \rightarrow \alpha \text{list}
\]

tl (tail) takes a list of alphas and returns a list of alphas (all elements of the list except the first one). Note that head and tail will raise an exception if called on the empty list \([],\).

\[
\text{length}: \alpha \text{list} \rightarrow \text{int}
\]

length takes a list of alphas and returns an integer.

\[
\text{map}: (\alpha \rightarrow \beta) \rightarrow \alpha \text{list} \rightarrow \beta \text{list}
\]

This is the map function in Google’s map-reduce [69]. map takes an apply, a list of alphas and returns a list of betas. Using the function with \(\alpha = \beta\) is possible, but having the type signature using \(\alpha\) and \(\beta\) makes the function more generic.

\[
\text{fold}: (\alpha \rightarrow \beta \rightarrow \alpha) \rightarrow \alpha \rightarrow \beta \text{list} \rightarrow \alpha
\]

The reduce in Google’s map-reduce [69] is a kind of fold. fold takes an accumulator, an \(\alpha\), a list of betas and returns an \(\alpha\).

\[
\text{iter}: (\alpha \rightarrow \text{unit}) \rightarrow \alpha \text{list} \rightarrow \text{unit}
\]

iter (iterate) takes a side-effect, a list of alphas and returns nothing.

\[
\text{exists}: (\alpha \rightarrow \text{bool}) \rightarrow \alpha \text{list} \rightarrow \text{bool}
\]

exists takes a predicate, a list of alphas and returns a Boolean.

\[
\text{filter}: (\alpha \rightarrow \text{bool}) \rightarrow \alpha \text{list} \rightarrow \alpha \text{list}
\]

filter takes a predicate, a list of alphas and returns a list of alphas (the ones satisfying the predicate).

\[
\text{partition}: (\alpha \rightarrow \text{bool}) \rightarrow \alpha \text{list} \rightarrow \alpha \text{list} \ast \alpha \text{list}
\]

partition takes a predicate, a list of alphas and returns a pair of list of alphas (elements satisfying the predicate on the left, others on the right).

\[
\text{sort}: (\alpha \rightarrow \alpha \rightarrow \text{int}) \rightarrow \alpha \text{list} \rightarrow \alpha \text{list}
\]

sort takes a comparison, a list of alphas and returns a list of alphas (sorted according to the order defined by the comparison function).

Programming most parts of the list module from scratch is an excellent exercise for any student of the language.

An OCaml programming environment

Here follows a selection of tools for OCaml programming in a UNIX-like environment. While different users may use different tools, some of them are quite standard in a productive and modern development environment.

OPAM  the OCaml Package Manager [70] allows to automatically install OCaml software, libraries (Fig. 2) and their dependencies (even system ones). OPAM is a source-based, user-level package manager. It can install a given compiler version and packages in a so-called “switch”, under the user’s home directory. The collection of open source OPAM packages is maintained by the community [71].

opam-bundle  can create a stand-alone, self-extracting and automatic installer for any OCaml software with an OPAM package description file [72].

utop  utop [73] is an improved top-level (interactive interpreter). Utop supports line editing, history, automatic completion, colorful syntax highlighting and more. Utop can be controlled within Emacs or as a standalone
terminal application. In the Python world, the equivalent of utop would be ipython.

Merlin is an editor helper [74]. It provides completion, type information and source browsing (jump to definition/list uses) for Vim and Emacs. Thanks to Merlin, standard editors become full integrated development environments for OCaml.

Emacs with modes like tuareg, ocaml or merlin, writing OCaml programs under Emacs is productive. Vim also has good support for OCaml. Microsoft Visual Studio Code [75] and Atom [76] also have some support for OCaml.

Dune is the best choice to manage the compilation of OCaml projects. It is very fast, has no system dependencies and supports parallel builds on all platforms. Build descriptions are terse but still human-readable (see Fig. 3).

ocp-browser is a terminal program to browse the interface and documentation of all installed OCaml libraries in an OPAM switch. ocp-browser alleviates the need to search and read HTML documentation online while programming.

ocp-indent & ocamlformat automate and standardize the indentation of OCaml source code. ocp-indent [77] and ocamlformat [78] integrate well with Emacs and Vim.

Results

OCaml in chemoinformatics and structural bioinformatics

We list some open source OCaml software that resulted from research in chemoinformatics and structural bioinformatics [79].

The bisector-tree data-structure described in the introduction is not a toy example. It can be used to accelerate similarity searches (Fig. 4).

For ligand-based virtual screening in 3D, the AutoCorrelation of Partial Charges method (ACPC [28]) uses the autocorrelation function [81] and linear binning [82] to encode all atoms of a molecule into a rotation-translation invariant representation. ACPC allows to rank-order a database of compounds versus a query molecule and was released in open source (opam package acpc [83]). ACPC performed remarkably well in retrospective ligand-based
virtual screening experiments. At an average speed of 1649 molecule/s, ACPC reached an average median area under the curve of 0.81 on 40 Directory of Useful Decoys [84] targets.

Consent [14, 85] (opam package lbvs_consent [86]) performs ligand-based virtual screening using consensus queries. When several active molecules are known, screening with all of them is recommended (instead of using just one). A consensus query can be created by screening serially with different ligands before merging similarity scores, or by combining chemical fingerprints. Consent was tested on 19 protein targets, 3776 known active and \( \sim 2 \times 10^6 \) inactive molecules from high throughput screening datasets. Three fingerprints were investigated (MACCS, ECFP4 and an unfolded fingerprint). Different consensus policies and consensus sizes (number of known actives) were benchmarked. A consensus fingerprint is always faster. In some circumstances, it can approach the performance of a consensus of scores in terms of Area Under the Receiver Operating Characteristic (ROC) Curve (AUC) and early retrieval.

EleKit [26, 27] was the first structural bioinformatics software able to measure the similarity of a ligand’s electrostatic field with that of a protein binding at a protein-protein interface (Fig. 5). Ligands showing a high similarity in this setting are potential drugs breaking protein-protein interactions. EleKit was a complex software, driving PDB2PQR [87], parsing PQR files, running the Adaptive Poisson-Boltzmann Solver (APBS [88]) in parallel, parsing ABPS output files, creating and operating 3D Boolean masks.

Also in structural bioinformatics, Fragger [25, 89, 90] is a protein fragment picker for 3D structural queries. From a set of PDB files, Fragger can create a protein fragments database. All fragment lengths are supported. Using the triangular inequality, Fragger can efficiently search with a query fragment and a distance threshold. Matching fragments are ranked by distance to the query, which can contain structural gaps. The allowed amino acid sequences matching a query can be constrained. Fragger is meant for protein design, loop grafting and related activities.
OCaml executables are fast. In terms of speed, OCaml is placed just after Go in the Debian language shootout [91]; the fastest language being C++ then C. However, execution speed is not the most important in a research setting. Programmer productivity is more important. In terms of verbosity, OCaml code is close to Python and far from Java (see Fig. 6). From past experience, an AUC calculation in OCaml is about 20 times faster than the equivalent python script [92]. While performing an AUC calculation faster may not seem important, to scientifically validate a computational method, one might run thousands of such calculations.

Since molecules can be processed independently, most chemoinformatics tasks are easy to parallelize. The Parmap OCaml library [94] provides parallel iter, map and fold functions for arrays and lists on multi-core computers. Parallelizing code with Parmap is trivial (Fig. 7). Parmap preserves semantics while achieving nearly optimal speedup [94] (Figs. 8, 9).

For stream computing, when a program cannot hold all items in memory (which is required by Parmap), we developed the parany library (opam package parany).
Parany is more generic than parmap. It is structured around three functions. An unfold function called demux, an apply function called work and a fold/reduce function called mux.

\[\text{demux} : \text{unit} \rightarrow \alpha\]
\[\text{work} : \alpha \rightarrow \beta\]
\[\text{mux} : \beta \rightarrow \text{unit}\]

Some more complex technologies exist to write even higher performance OCaml programs. SPOC [96] is an OCaml library allowing general purpose GPU programming, using Cuda or OpenCL kernels. SPOC allows to create specific data sets usable by those kernels and automatically manages memory transfers between CPU and GPU.

BER MetaOCaml [97] is an OCaml dialect for multi-stage programming [98]. It allows run-time C code generation and program execution. BER MetaOCaml can be used to compile domain-specific languages and automate the specialization of high-performance computational kernels.
Discussion
Scientific software prototyping in OCaml

In an academic research setting, it is common for a software project to be severely under staffed, compared to industrial standards, i.e. a single person might be in charge of the full software life-cycle (requirements gathering, specification, design, implementation, speed optimization, parameter tuning, test and validation, release and packaging, maintenance). In research, requirements are ill-defined and changing. Since the purpose of the software is to scientifically show that an idea works, having a high confidence in the software is important. Moreover, during the course of the project, design decisions might change and impact the whole code-base. OCaml types and compiler allow to refactor software fast and without missing any place that needs changing (see Fig. 10 for an example of refactoring that only took a few minutes). Thus, during prototyping, the programmer is not afraid to do drastic changes to the software (agility). In such a setting, and when using OCaml, we propose to abandon the practice of unit tests. Because, there is not enough manpower to write and maintain them. Note however that OCaml has tools for programmers who want to write unit [99] or property-based tests [100] as comments inside their code. Since the software will change a lot during its lifetime, maintaining unit tests would be too costly and slow down the pace of research. Of course, if we were using a dynamically-typed language such as Python or Ruby, such a decision would be risky and many problems discovered at run-time. Instead of unit tests, we propose to use regression tests and end-to-end validation, once a prototype is advanced enough. For example, a valid output can be verified by hand from a known input and added to a set of regression tests.

In the same vein, we propose to abandon OCaml interface files when prototyping. Having to maintain interface files slows down refactoring. Interface files of libraries should only be added once a project is going to be released.

When programming in OCaml, one strongly relies on the compiler to catch errors. It is common to see a complex but compiling OCaml program run without any run-time error, even when running for the first time. Programs written in Haskell have this exact same property.

OCaml language and ecosystem drawbacks

When working in OCaml, if functors and module signatures are heavily used, compiler error messages can become hard to understand. Also, the required syntax is nontrivial and might need some practice.

For chemoinformatics, a parser for Simplified Molecular-Input Line-Entry System (SMILES [101]) and a parser for SMiles ARbitrary Target Specification (SMARTS [102]) are the most obvious missing libraries. Also, nowadays it would not be reasonable to do chemoinformatics research without using the functionalities of the Chemistry Development Kit (CDK [103, 104]), Rdkit [105] or Open Babel [106]. Since there are no OCaml bindings to those libraries, our current solution is to write small programs interfacing with them, in order to extract or import data to/from them. By following the UNIX design principles [107], it is easy to create, debug and maintain software that exchange data via text files. However, in some projects [14, 26, 28], we have written parsers for parts of the PDB [108], PQR [87] and MOL2 [109] file formats.

Currently, the OCaml ecosystem is weak in the Machine Learning field, especially when compared to Python and the Scikit-learn [110] library. At least, there is one library for classification using random forests [111] (opam package orandforest [112]) and a numerical library (opam package owl [113, 114]) with some machine learning functionalities like regression and neural networks. For deep learning, some OCaml bindings to TensorFlow [115, 116] and PyTorch [117] have been released recently. To palliate the deficiency in machine learning libraries, we have recently developed several OCaml packages tapping into the R [118] ecosystem; for support vector machines (opam package orsvm-e1071 [119]), random forests (opam package orrandomForest [120]) and gradient boosted trees (opam package orxgboost [102]). We have also developed the classification performance metrics library in order to benchmark virtual screening experiments (opam package cpmlib [121]). Cpmlib features ROC curves, AUC [122], enrichment factor, power
metric [123] and Boltzmann-Enhanced Discrimination of ROC (BEDROC [124]).

OCaml is best for back-end and system [125] programming. To quickly annotate molecules or protein structures, rather than doing graphics programming in OCaml, we recommend generating BILD [126] files. BILD files are simple, human-readable line-oriented text files, easy to generate by a program or by hand. They can be viewed within UCSF Chimera [127] (Fig. 11).

While OCaml is a portable language, not all programmers write portable programs. OCaml code can be automatically translated to JavaScript [128] to target web browsers (opam package js_of_ocaml). But parallel programs or programs relying extensively on the Unix module might not work under Windows. Also, there may be less libraries/opam packages available under Windows. If Windows support is a primary concern, F# or Haskell [16] might be safer programming language choices. If access to a comprehensive cheminformatics library is a prime concern, Scala might be a safer choice since its interoperability with Java would allow using the Chemistry Development Kit.

For managers, the fact that there are few OCaml programmers available on the market is a concern. However, we feel that programmers can become proficient in the language quickly, so this is not a major concern.

Conclusions
OCaml is a strongly typed programming language of the functional family. In this article, we have tried to share our experience in using it for Chemoinformatics and Structural Bioinformatics research.

This article should not be seen as an attempt at asserting the superiority of OCaml and/or functional programming over other programming languages and approaches. Rather, we encourage researchers to choose and use the tools that make them the most productive, even if those tools are not mainstream.

To us, OCaml has been proven quite productive for software prototyping in Chemoinformatics and Structural Bioinformatics method development. The software demonstrated here were used intensively and timely during scientific validation campaigns, on many molecules and protein targets. We have never regretted our choice of OCaml and still use it today.

Authors’ contributions
FB wrote the software, ran computational experiments and prepared all figures and tables. All authors read and approved the final manuscript.

Author details
1 Department of Bioscience and Bioinformatics, Faculty of Computer Science and Systems Engineering, Kyushu Institute of Technology, Iizuka, Fukuoka, Japan. 2 Laboratory for Structural Bioinformatics, Center for Biosystems Dynamics Research, RIKEN, Yokohama, Kanagawa, Japan. 3 PRESTO, Japan Science and Technology Agency, Kawaguchi, Saitama, Japan.

Acknowledgements
FB is a JSPS international research fellow http://www.jsps.go.jp/english. This work was supported by JST PRESTO Grant Number JPMJPR15D8 and JSPS KAKENHI Grant Numbers 18H03334 and 18H02395. FB acknowledges the use of ChemAxon’s JChem (2017) http://www.chemaxon.com. All authors acknowledge the use of Omega 2.5.1.4 from OpenEye Scientific Software, Santa Fe, NM http://www.eyesopen.com. Some of the computing power used in this study was provided by RIKEN ACCC, on the Hokusai Large Memory Application Computing Server. FB thanks all participants in the OCaml open-source ecosystem for the many excellent libraries, tools and user support.

Competing interests
The authors declare that they have no competing interests.

Consent for publication
Not applicable.

Ethics approval and consent to participate
Not applicable.

Publisher’s Note
Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Received: 6 September 2018 Accepted: 22 January 2019
Published online: 05 February 2019

References
1. Colmerauer A, Roussel P (1996) The birth of Prolog. In: History of programming languages—II. ACM, New York, pp 331–367. https://doi.org/10.1145/234286.1057820
2. Hughes J (1989) Why functional programming matters. Comput J 32(2):98–107. https://doi.org/10.1093/comjnl/32.2.98
3. Hudak P (1994) Haskell vs Ada vs C++ vs Awk vs... an experiment in software prototyping productivity. Contract 14(92–C):0153
4. Wiger U (2001) Four-fold increase in productivity and quality—industrial-strength functional programming in telecom-class products. Ericsson Telecom Ab, Stockholm
5. Pavel Y (2018) Full support of OpenSMILES specification for Haskell. http://github.com/znatech/smiles. Accessed 2018-12-01
6. A, JC (2018) OpenSMILES specification version 1.0, 2016-05-15. http://opensmiles.org/opensmiles.html. Accessed 2018-12-01
7. Krzysztof L (2018) Haskell library for chemistry. http://github.com/xlanger/radium. Accessed 2018-12-01
8. Stefan H (2018) Purely functional cheminformatics toolkit written in Scala. http://github.com/stefan-hoeck/chemf. Accessed 2018-12-01
9. Höck S, Riedl R (2012) chemf: a purely functional chemistry toolkit. J Cheminform 4(1):38. https://doi.org/10.1186/1758-2946-4-38
10. Leroy X, Doligez D, Frisch A, Garrigue J, Rémy D et al (2016) The OCaml system release 4.04: Documentation and user’s manual, Inria. https://hal.inria.fr/hal-00902013v3/document
11. Hindley R (1969) The principal type-scheme of an object in combinatory logic. Trans Am Math Soc 146:29–60
12. Milner R (1978) A theory of type polymorphism in programming. J Comput Syst Sci 17(3):348–375. https://doi.org/10.1016/0022-0000(78)90014-4
13. Pierce BC (2002) Types and programming languages. MIT press, Cambridge
14. Berenger F, Vu O, Meiler J (2017) Consensus queries in ligand-based virtual screening experiments. J Cheminform 9(1):60. https://doi.org/10.1186/s13321-017-0248-5
15. Wadler P (1990) Comprehending monads. In: Proceedings of the 1990 ACM conference on LISP and functional programming, LFP ’90. ACM, New York, pp 61–78. https://doi.org/10.1145/91556.91592 (ISBN: 0-89791-368-X)
16. Peyton Jones SL (2003) Haskell 98: introduction. J Funct Program 13(1–2):6. https://doi.org/10.1017/S0956796803000315
17. Barras B, Boutin S, Cornes C, Courant J, Filliatre J-C, Gimenez E, Herbelin H, Huet G, Munoz C, Murthy C et al (1997) The Coq proof assistant reference manual version 6.1. INRIA, Paris
18. Brady E et al (2008) Idris, a language with dependent types. In: IFL 2008
19. Brady E (2017) Type-driven development with Idris. Manning Publications, Shelter Island
20. Norell U (2009) Dependently typed programming in Agda. In: Koopman PFW, Plasmeijer R, Swierstra SD (eds) 6th international school on advanced functional programming, AFP 2008. Lecture notes in computer science, vol 5832. Springer, Berlin, Heidelberg, pp 230–266. https://doi.org/10.1007/978-3-642-04652-0_5
21. Mondet S, Aksoy BA, Rozenberg L, Hodes I, Hammerbacher J (2017) Bioinformatics workflow management with the Wobidisco ecosystem. https://doi.org/10.1016/j.tice.2018.04.001
22. Rubinstein Y, Kodysh J, Hodes I, Mondet S, Aksoy BA, Finnigan JP, Bhardwaj N, Hammerbacher J (2017) Computational pipeline for the PGV-001 neoantigen vaccine trial. https://doi.org/10.1101/174516
23. Rozenberg L, Hammerbacher J (2018) Prohlotype: a probabilistic framework for HLA typing. https://doi.org/10.1101/244962
24. Jambon M, Andrieu O, Combet C, Deléage G, Delfaud F, Geourjon C (2005) The Sulfo server. 3D search for protein functional sites. Bioinformatics 21(39):3929–3930. https://doi.org/10.1093/bioinformatics/bti645
25. Berenger F, Simoncini D, Voet A, Shestha R, Zhang KYJ (2018) Framer: a protein fragment picker for structural queries [version 2; referees 2 approved]. F1000Research 6(1223). https://doi.org/10.12688/f1000research.12486.2
26. Voet A, Berenger F, Zhang KYJ (2013) Electrostatic similarities between protein and small molecule ligands facilitate the design of protein–protein interaction inhibitors. PLoS ONE 8(10):1–9. https://doi.org/10.1371/journal.pone.0075762
27. Voet ARD, Kumar A, Berenger F, Zhang KYJ (2014) Combining in silico and in silico approaches for virtual screening and pose prediction in SAMPL4. J Comput Aid Mol Des 28(4):363–373. https://doi.org/10.1007/s10822-013-9702-2
28. Berenger F, Voet A, Lee XY, Zhang KYJ (2014) A rotation-translation invariant molecular descriptor of partial charges and its use in ligand-based virtual screening. J Cheminform 6(1):23. https://doi.org/10.1039/c3cj00047k
29. Feret J, Danos V, Krivine J, Fontana W (2009) Interaction networks in the absence of run-time errors and data races with Astrée. In: Eighth European congress on embedded real time software and systems, Toulouse, France
30. Ball T, Rajamani SK (2002) The slam project: debugging system software via static analysis. In: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on principles of programming languages, POPL ’02. ACM, New York, NY, USA, pp 1–3. https://doi.org/10.1145/503271.503274
31. Pierce BC, Vuillon J (2004) What’s in unison? A formal specification and reference implementation of a file synchronizer. Technical report MS-CIS-03-36, Department of Computer and Information Science, University of Pennsylvania
32. Le Fessant F, Patarin S (2003) MLdonkey, a multi-network peer-to-peer file-sharing program. Research report RR-4797. INRIA
33. INRIA (2018) The Coq proof assistant. http://coq.inria.fr. Accessed 2018-12-01
34. Frigo M (1999) A fast Fourier transform compiler. In: Proceedings of the ACM SIGPLAN 1999 conference on programming language design and implementation, PLDI ’99, vol 34. ACM, New York, pp 169–180. https://doi.org/10.1145/301618.301661
35. Chailoux E, Manouy P, Pagano B (2007) Développement d’applications avec Objective Caml. O’REILLY & Associates, France. https://caml.inria.fr/pub/docs/oreilly-book-oocaml-ora-book.pdf (ISBN: 2-84177-121-0)
36. Minsky Y, Madhavapeddy A, Hickey J (2013) Real World OCaml: functional programming for the masses. O’Reilly Media Inc, Sebastopol
37. Whittington J (2013) OCaml from the very beginning. Coherent Press, Birmingham
38. Emmanuel C, Pascal M, Bruno P (2018) Developing applications with objective Caml. http://caml.inria.fr/pub/docs/oreilly-book.html. Accessed 2018-12-01
39. Xavier L, Didier R (2018) Unix system programming in OCaml. http://ocaml.github.io/ocamlunix. Accessed 2018-12-01
40. Lipovaca M (2011) Learn you a Haskell for great good! No Starch Press, San Francisco
41. Abelhson H, Sussman GJ, Sussman J (1996) Structure and interpretation of computer programs, 2nd edn. The MIT Press, Cambridge
42. Yaron M (2018) Caml trading. www.youtube.com/watch?v=hKcOkWz0j_s. Accessed 2018-12-01
43. Minsky Y, Weeks S (2008) Caml trading—experiences with functional programming on wall street. J Funct Program 18(4):553–564. https://doi.org/10.1145/1675357.1675361
44. Minsky Y, Madhavapeddy A, Hickey J (2018) Real World OCaml. http://v1.realworldocaml.org/v1/en.html. Accessed 2018-12-01
45. Xavier L, Didier R (2018) Unix system programming in OCaml. http://ocaml.github.io/ocamlunix. Accessed 2018-12-01
46. Simon C (2018) OCaml-containers. http://github.com/c-cube/ocaml-containers. Accessed 2018-12-01
47. Minsky Y, Weeks S (2008) Caml trading—experiences with functional programming on wall street. J Funct Program 18(4):553–564. https://doi.org/10.1145/1675357.1675361
48. Berg M, Cheong O, Kreveld M, Overmars M (2008) Computational geometry: algorithms and applications, 3rd edn. Springer, Santa Clara
49. INRIA (2018) CAML Consortium. http://caml.inria.fr/camlconsortium. Accessed 2018-12-01
50. Simon C (2018) OCaml-containers documentation. http://simon.cedelea.fr/ocaml-containers/fast/containers/index.html. Accessed 2018-12-01
51. community O (2018) OCaml batteries included. https://github.com/ocaml-batteries-team/batteries-included. Accessed 2018-12-01
52. Simon C (2018) OCaml-batteries team/batteries-included. http://ocaml-batteries-team.github.io/batteries-included/hdoc2. Accessed 2018-12-01
53. Stroud J (2018) Jane street core documentation. http://ocaml.janestreet.com. Accessed 2018-12-01
54. Simon C (2018) OCaml-batteries team/batteries-included/hdoc2. http://ocaml.batteries-included/hdoc2. Accessed 2018-12-01
55. Simon C (2018) OCaml-containers documentation. http://simon.cedelea.fr/ocaml-containers/fast/containers/index.html. Accessed 2018-12-01
56. community O (2018) OCaml batteries included. https://github.com/ocaml-batteries-team/batteries-included. Accessed 2018-12-01
57. Simon C (2018) OCaml-containers documentation. http://simon.cedelea.fr/ocaml-containers/fast/containers/index.html. Accessed 2018-12-01
58. Berg M, Cheong O, Kreveld M, Overmars M (2008) Computational geometry: algorithms and applications, 3rd edn. Springer, Santa Clara
117. Mazare L (2018) ocaml-torch. http://github.com/LaurentMazare/ocaml-torch. Accessed 2018-12-01
118. R Core Team (2018) R: a language and environment for statistical computing. R Foundation for Statistical Computing, Vienna, Austria
119. Berenger F (2018) orsvm_e1071 - OCaml wrapper to R packages e1071 and sympath. http://github.com/UnixJunkie/orsvm-e1071. Accessed 2018-12-01
120. Berenger F (2018) orrandomForest—classification or regression using random forests. http://github.com/UnixJunkie/orrandomForest. Accessed 2018-12-01
121. Berenger F (2018) cpml—classification performance metrics library. http://github.com/UnixJunkie/cpmlib. Accessed 2018-12-01
122. Bradley AP (1997) The use of the area under the ROC curve in the evaluation of machine learning algorithms. Pattern Recogn 30(7):1145–1159. https://doi.org/10.1016/S0031-3203(96)00142-2
123. Lopes JCD, dos Santos FM, Martins-José A, Augustyns K, De Winter H (2017) The power metric: a new statistically robust enrichment-type metric for virtual screening applications with early recovery capability. J Cheminform 9(1). https://doi.org/10.1186/s13321-016-0189-4
124. Truchon J-F, Bayly CI (2007) Evaluating virtual screening methods: good and bad metrics for the early recognition problem. J Chem Inf Model 47(2):488–508. https://doi.org/10.1021/ci600426e
125. Leroy X, Rémy D (2014) Unix system programming in OCaml
126. UCSF (2018) Chimera BILD file format. http://www.cgl.ucsf.edu/chimera/docs/UsersGuide/bild.html. Accessed 2018-12-01
127. Petterson EF, Goddard TD, Huang CC, Couch GS, Greenblatt DM, Meng EC, Ferrin TE (2004) UCSF Chimera—a visualization system for exploratory research and analysis. J Comput Chem 25(13):1605–1612. https://doi.org/10.1002/jcc.20084
128. Vouillon J, Balat V (2014) From bytecode to JavaScript: the Js of ocaml compiler. Softw Pract Exp 44(8):951–972. https://doi.org/10.1002/spe.2187