Berenger *et al. J Cheminform* (2019) 11:10
https://doi.org/10.1186/s13321-019-0332-0

Journal of Cheminformatics

RESEARCH ARTICLE Open Access

Chemoinformatics and structural bioinformatics in OCaml

Francois Berenger[1*], Kam Y. J. Zhang[2] and Yoshihiro Yamanishi[1,3]

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^{f}' 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 $\sim$ 3000 OCaml lines (without comments) of the consent software [14] and its four executables compile from scratch and link in $\sim$ 3.8 s (resp $\sim$ 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, OCamlPRO 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.

```
type bucket = { vp: P.t; (* vantage point *)
                sup: float; (* max dist to vp among bucket points *)
                points: P.t array } (* remaining points (vp excluded) *)
type node =
  { (* left half-space *)
    l_vp: P.t; (* left vantage point *)
    l_sup: float; (* max dist to l_vp among points in same half-space *)
    (* right half-space *)
    r_vp: P.t; (* right vantage point *)
    r_sup: float; (* max dist to r_vp among points in same half-space *)
    (* sub-trees *)
    left: t;
    right: t }
and t = Empty
      | Node of node
      | Bucket of bucket
```

Fig. 1 OCaml code defining a bucketized bisector-tree. The code is parameterized by a point type (P.t). The implementation works with any point type, as long as it defines a distance function

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. α and β are standard names for polymorphic types.

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

Definition 1 The syntax

$$\texttt{apply} : \alpha \rightarrow \beta$$

defines the type of a function named `apply` from type α to type β in which α and β are type parameters. The equivalent C++ header file portion would be

```
template <class A, class B>
B apply (A a);
```

Definition 2 Let's call `accumulate` any function which takes an α, a β and returns an α.

$$\texttt{accumulate} : \alpha \rightarrow \beta \rightarrow \alpha$$

Definition 3 Let's call `side-effect` any function which takes an α and returns nothing (in OCaml, nothing's type is called unit).

$$\texttt{side-effect} : \alpha \rightarrow \mathit{unit}$$

Definition 4 Let's call `predicate` any function which takes an α and returns a Boolean.

`predicate` : $\alpha \rightarrow bool$

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

`comparison` : $\alpha \rightarrow \alpha \rightarrow int$

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

`cons`: $\alpha \rightarrow \alpha\ list \rightarrow \alpha\ list$

The `cons` (construct) function takes an α, 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 $\alpha = int$.

`hd`: $\alpha\ list \rightarrow \alpha$

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

`tl`: $\alpha\ list \rightarrow \alpha\ list$

`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 `[]`.

`length`: $\alpha\ list \rightarrow int$

`length` takes a list of alphas and returns an integer.

`map`: $(\alpha \rightarrow \beta) \rightarrow \alpha\ list \rightarrow \beta\ list$

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 α and β makes the function more generic.

`fold`: $(\alpha \rightarrow \beta \rightarrow \alpha) \rightarrow \alpha \rightarrow \beta\ list \rightarrow \alpha$

The reduce in Google's map-reduce [69] is a kind of fold. `fold` takes an accumulator, an α, a list of betas and returns an α.

`iter`: $(\alpha \rightarrow unit) \rightarrow \alpha\ list \rightarrow unit$

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

`exists`: $(\alpha \rightarrow bool) \rightarrow \alpha\ list \rightarrow bool$

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

`filter`: $(\alpha \rightarrow bool) \rightarrow \alpha\ list \rightarrow \alpha\ list$

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

`partition`: $(\alpha \rightarrow bool) \rightarrow \alpha\ list \rightarrow \alpha\ list * \alpha\ list$

`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).

`sort`: $(\alpha \rightarrow \alpha \rightarrow int) \rightarrow \alpha\ list \rightarrow \alpha\ list$

`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

```
opam-version: "2.0"
synopsis: "Bisector tree OCaml library"
description: """
<long-textual-description>
"""
maintainer: "<email-address>"
authors: "Francois Berenger"
license: "BSD-3"
homepage: "<long-URL>"
bug-reports: "<long-URL>"
dev-repo: "<long-URL>"
depends: [
  "ocaml"
  "batteries"
  "dune" {build}
]
build: ["dune" "build" "-p" name "-j" jobs]
url {
  src: "<long-URL>"
  checksum: "md5=<checksum>"
}
```

Fig. 2 OPAM package description file for the bisector tree library. Such a file allows OPAM to automatically install/uninstall from source this library and all its transitive dependencies

```
(library
  (name        bst)
  (public_name bst)
  (modules bisec_tree)
  (libraries batteries))

(executables
  (names     test)
  (modules   test)
  (libraries unix dolog batteries minicli
             bst))
```

Fig. 3 Complete build description file for the bisector tree library and it's test executable

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

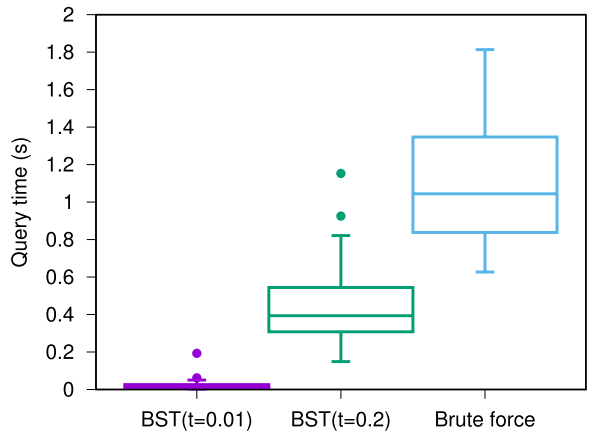

Fig. 4 Using a bisector-tree (BST) to accelerate similarity searches on a database of 10^6 PUBCHEM molecules. Molecular encoding is Faulon's signature molecular descriptor [80] (an unfolded-counted fingerprint) with height equal to one bond and parametrized over MOL2 atom types. The database is searched for all molecules with Tanimoto to query ≥ 0.99 (left; $t = 0.01$) or Tanimoto ≥ 0.8 (middle; $t = 0.2$). The brute force version is shown on the right. 50 molecules from the database were selected randomly to serve as queries. Creating the BST (database indexing) took approximately 5 min using a single core of our desktop computer

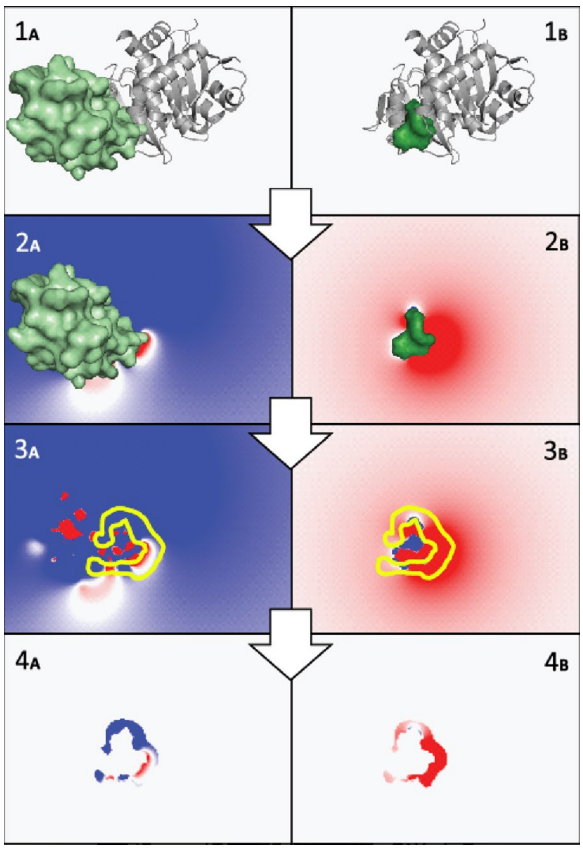

Fig. 5 Overview of EleKit applied to PDB codes 2B4J (1_A) and 3LPU (1_B). The "ligand-protein" is shown as a green surface in 1_A and 2_A. The "ligand-small-molecule" is shown as a smaller green surface in 1_B and 2_B. The receptor protein is shown as a gray cartoon in 1_A and 1_B. Electrostatic potential fields are calculated and stored in distinct grids (2_A and 2_B). A boolean mask in 3D is created to select the solvent region nearby the interface (3_A and 3_B). Finally, the similarity between electrostatic potentials in the masked region (4_A and 4_B) is calculated using the Spearman rank correlation coefficient (figure adapted from Voet [26])

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.

```
/* Java */
public class HelloWorld {
    public static void main (String[] args) {
        System.out.println("Hello, world!");
    }
}
// C++
#include <iostream>
int main() {
  std::cout << "Hello, World!\n";
}
(* OCaml *)
print_endline "Hello, world!"
# Python-2.7
print "Hello, world!"
```

Fig. 6 Valid hello world programs to illustrate the *idiomatic* verbosity of Java, C++, OCaml and Python-2.7. Keep in mind that in many programming languages, programmers can make their source code arbitrarily small, sometimes to the point that a program is no more readable

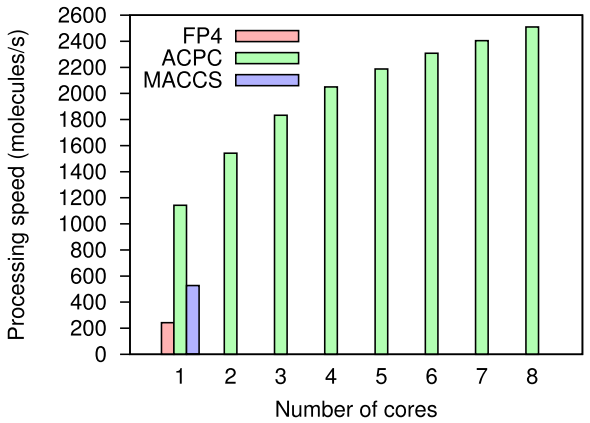

Fig. 8 Performance of ACPC in the electrostatic space, using Parmap for parallelization. Open Babel 2.3.9's MACCS and FP4 C++ implementations run-times are shown to give an order of magnitude. Run-times were averaged over three runs. Protein target: Human immunodeficiency virus type 1 protease (HIVPR) from the Database of Useful Decoys Enhanced (DUDE [93]); 26450 ligands and decoys

```
   let completed = open_out out_files in
   (* molecules from a multiple molecules mol2
      file are processed in parallel *)
-  L.iter
+  Parmap.pariter ~ncores:nprocs
     (fun (pdb_B', m_name) ->
        P.printf "creating .pqr for %s...\n%!"
          pdb_B';
        let maybe_pqr_B =
          [...]
          let dx_out_B_gz = MU.gzip dx_out_B 1 in
          P.fprintf completed "%s %s %s\n%!"
            dx_out_B_gz mask_file_B m_name)
-    molecules;
+    (Parmap.L molecules);
   close_out completed
```

Fig. 7 Git diff after parallelization of EleKit. Parallelizing EleKit using Parmap was a two lines change in ∼ 3000 lines of code. All program development and debugging was done on sequential code. With an electrostatic calculation run-time of approximately 2 min per small molecule, parallelization was mandatory for production use of EleKit on thousands of molecules

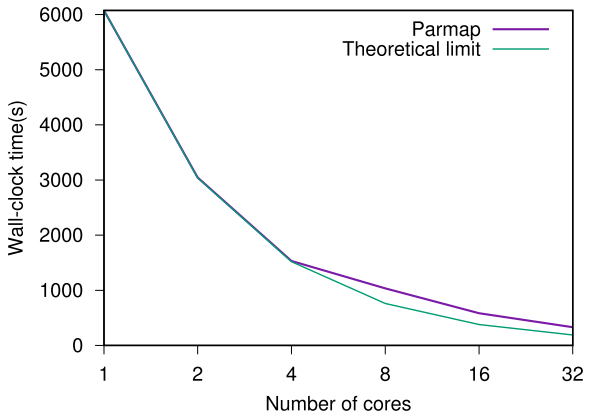

Fig. 9 Wall-clock time to analyze hundreds of molecules with EleKit and Parmap. Up to four cores, the parallelization performance is almost indistinguishable from a perfectly parallelizing program (theoretical limit)

Accelerating chemoinformatics and structural bioinformatics in OCaml

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

```
--- a/svm_common.ml
+++ b/svm_common.ml

let optimal_lambda nb_features ncores train test =
   [...]
   let best_lambda, best_auc =
     L.fold_left (fun (best_lambda, best_auc) (lambda, auc) ->
         if auc > best_auc then (lambda, auc)
         else (best_lambda, best_auc)
       ) (0.0, 0.0) lambda_aucs in
-  (best_lambda, extract_fn model)
+  if best_auc <= 0.5 then
+    let () = Log.warn "AUC not improved upon lambda scan; model discarded" in
+    None
+  else
+    let () = Log.info "best_lambda: %f best_AUC: %.3f" best_lambda best_auc in
+    Some (best_lambda, extract_fn model)

let train_one nb_features ncores train_bag =
   match Common.train_test_split (Train_portion 0.8) train_bag with
   | Train_test (train, test) ->
-    let lambda, _model = optimal_lambda nb_features ncores train test in
-    Log.info "keeping lambda but retraining on whole bag";
-    let model = svmpath_train nb_features train_bag in
-    (lambda, extract_fn model)
+    begin
+      match optimal_lambda nb_features ncores train test with
+      | None -> None
+      | Some (lambda, _model) ->
+        Log.info "keeping lambda but retraining on whole bag";
+        let model = svmpath_train nb_features train_bag in
+        Some (lambda, extract_fn model)
+    end
     [...]
--- a/svm_train.ml
+++ b/svm_train.ml

    let train_bags = Bagger.bags rng nb_bags train_mols in
    Log.info "training all models";
-   let models = L.map (SVM_common.train_one nb_features ncores) train_bags in
+   let models =
+     L.fold_left (fun acc train_bag ->
+         match SVM_common.train_one nb_features ncores train_bag with
+         | Some x -> x :: acc
+         | None -> acc
+       ) [] train_bags in
```

Fig. 10 Git diff excerpt of an actual code refactoring in the SVM part of a category-QSAR software. Sometimes, the R svmpath package encounters numerical problems, like an exactly or computationally singular matrix. To deal with such rare cases, it was decided to drop a model from the bag of models. Since a bagging classifier with 21 models was being trained, dropping one or two models was deemed better than letting the whole software crash. Hence, an option type was introduced in the function optimal_lambda from file 'svm_common.ml', along with proper warning messages. Then, the compiler forced updating the rest of the code

[95]). 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.

$$\texttt{demux} : unit \rightarrow \alpha$$
$$\texttt{work} : \alpha \rightarrow \beta$$
$$\texttt{mux} : \beta \rightarrow unit$$

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.

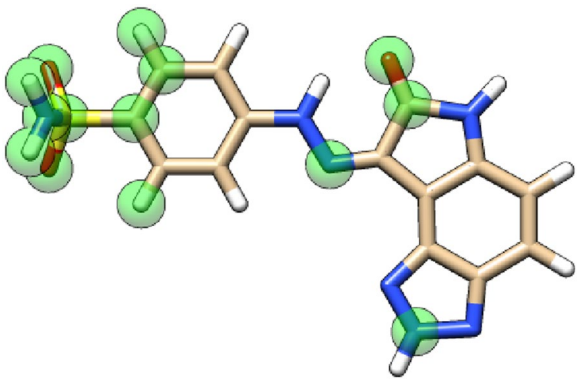

Fig. 11 Graphical annotation of a query molecule using a BILD file generated by the ACPC software for viewing with UCSF Chimera. A query molecule of the CDK2 protein target is annotated in the electrostatic space, based on atomic contributions to AUC. Transparent green balls highlight atoms which if masked (their contribution is removed from the molecular encoding/fingerprint) would decrease the AUC reached by this molecule in a similarity search

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 vain, 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 taping 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 chemoinformatics 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.

Acknowlegements

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

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/zmactep/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/klangner/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-00930213v3/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):0–6. 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 PWM, 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/10.1101/213884
22. Rubinsteyn A, 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) Prohlatype: 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 SuMo server: 3D search for protein functional sites. Bioinformatics 21(20):3929–3930. https://doi.org/10.1093/bioinformatics/bti645
25. Berenger F, Simoncini D, Voet A, Shrestha R, Zhang KYJ (2018) Fragger: a protein fragment picker for structural queries [version 2; referees: 2 approved]. F1000Research 6(1722). 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 cerebro approaches for virtual screening and pose prediction in SAMPL4. J Comput Aided 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/10.1186/1758-2946-6-23
29. Danos V, Feret J, Fontana W, Harmer R, Krivine J (2007) Rule-based modelling of cellular signalling. In: Caires L, Vasconcelos VT (eds) CONCUR 2007—concurrency theory. Springer, Berlin, pp 17–41
30. Feret J, Danos V, Krivine J, Harmer R, Fontana W (2009) Internal coarse-graining of molecular systems. Proc Natl Acad Sci 106(16):6453–6458. https://doi.org/10.1073/pnas.0809908106
31. Deeds EJ, Krivine J, Feret J, Danos V, Fontana W (2012) Combinatorial complexity and compositional drift in protein interaction networks. PLoS ONE 7(3):1–14. https://doi.org/10.1371/journal.pone.0032032
32. Boutillier P, Maasha M, Li X, Medina-Abarca HF, Krivine J, Feret J, Cristescu I, Forbes AG, Fontana W (2018) The Kappa platform for rule-based modeling. Bioinformatics 34(13):583–592. https://doi.org/10.1093/bioinformatics/bty272
33. Charles S, Veber P, Delignette-Muller ML (2018) MOSAIC: a web-interface for statistical analyses in ecotoxicology. Environ Sci Pollut Res 25(12):11295–11302. https://doi.org/10.1007/s11356-017-9809-4
34. INRIA (2018) Caml Consortium. http://caml.inria.fr/consortium. Accessed 2018-12-01
35. Calcagno C, Distefano D, Dubreil J, Gabi D, Hooimeijer P, Luca M, O'Hearn P, Papakonstantinou I, Purbrick J, Rodriguez D (2015) Moving fast with software verification. In: Havelund K, Holzmann G, Joshi R (eds) NASA formal methods. Springer, Cham, pp 3–11
36. Peyton Jones S, Eber J-M, Seward J (2000) Composing contracts: an adventure in financial engineering (functional pearl). In: Proceedings of the fifth ACM SIGPLAN international conference on functional programming. ICFP '00. ACM, New York, NY, USA, pp 280–292. https://doi.org/10.1145/351240.351267
37. Miné A, Mauborgne L, Rival X, Feret J, Cousot P, Kastner D, Wilhelm S, Ferdinand C (2016) Taking static analysis to the next level: proving the absence of run-time errors and data races with Astrée. In: Eighth European congress on embedded real time software and systems, Toulouse, France
38. 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/10.1145/503272.503274
39. Pierce BC, Vouillon 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
40. Le Fessant F, Patarin S (2003) MLdonkey, a multi-network peer-to-peer file-sharing program. Research report RR-4797. INRIA
41. INRIA (2018) The Coq proof assistant. http://coq.inria.fr. Accessed 2018-12-01
42. 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
43. Chailloux E, Manoury P, Pagano B (2007) Développement d'applications avec Objective Caml. O'REILLY & Associates, France. https://caml.inria.fr/pub/docs/oreilly-book/ocaml-ora-book.pdf **(ISBN: 2-84177-121-0)**
44. Minsky Y, Madhavapeddy A, Hickey J (2013) Real World OCaml: functional programming for the masses. O'Reilly Media Inc, Sebastopol
45. Whitington J (2013) OCaml from the very beginning. Coherent Press, Birmingham
46. 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
47. Minsky Y, Madhavapeddy A, Hickey J (2018) Real World OCaml. http://v1.realworldocaml.org/v1/en/html. Accessed 2018-12-01
48. Xavier L, Didier R (2018) Unix system programming in OCaml. http://ocaml.github.io/ocamlunix. Accessed 2018-12-01
49. Lipovaca M (2011) Learn you a Haskell for great good!. No Starch Press, San Francisco
50. Abelson H, Sussman GJ, Sussman J (1996) Structure and interpretation of computer programs, 2nd edn. The MIT Press, Cambridge
51. Yaron M (2018) Caml trading. www.youtube.com/watch?v=hKcOkWzj0_s. Accessed 2018-12-01
52. Minsky Y, Weeks S (2008) Caml trading—experiences with functional programming on wall street. J Funct Program 18(4):553–564. https://doi.org/10.1017/S095679680800676X
53. Minsky Y (2011) OCaml for the Masses. Commun ACM 54(11):53–58. https://doi.org/10.1145/2018396.2018413
54. OCamlPRO (2018) Try OCaml. http://try.ocamlpro.com. Accessed 2018-12-01
55. Xavier L, Damien D, Alain F, Jacques G, Didier R, Jérôme V (2018) The OCaml system release 4.07. https://caml.inria.fr/pub/docs/manual-ocaml. Accessed 2018-12-01
56. Xavier L, Damien D, Alain F, Jacques G, Didier R, Jérôme V (2018) The standard library. http://caml.inria.fr/pub/docs/manual-ocaml/stdlib.html. Accessed 2018-12-01
57. Simon C (2018) OCaml-containers. http://github.com/c-cube/ocaml-containers. Accessed 2018-12-01
58. Simon C (2018) OCaml-containers documentation. http://simon.cedeela.fr/ocaml-containers/last/containers/index.html. Accessed 2018-12-01
59. community O (2018) OCaml batteries included. https://github.com/ocaml-batteries-team/batteries-included. Accessed 2018-12-01
60. community, O (2018) Batteries user guide. http://ocaml-batteries-team.github.io/batteries-included/hdoc2. Accessed 2018-12-01
61. Street J (2018) Janestreet core. https://github.com/janestreet/core. Accessed 2018-12-01
62. Street J (2018) Jane street core documentation. http://ocaml.janestreet.com/ocaml-core/latest/doc/core. Accessed 2018-12-01
63. Kalantari I, McDonald G (1983) A data structure and an algorithm for the nearest point problem. IEEE Trans Softw Eng 5:631–634
64. Berg M, Cheong O, Kreveld M, Overmars M (2008) Computational geometry: algorithms and applications, 3rd edn. Springer, Santa Clara

65. Uhlmann JK (1991) Satisfying general proximity/similarity queries with metric trees. Inf Process Lett 40(4):175–179. https://doi.org/10.1016/0020-0190(91)90074-R
66. Yianilos PN (1993) Data structures and algorithms for nearest neighbor search in general metric spaces. In: Proceedings of the fourth annual ACM-SIAM symposium on discrete algorithms. SODA '93. SIAM, Philadelphia, pp 311–321
67. Xu H, Agrafiotis DK (2003) Nearest neighbor search in general metric spaces using a tree data structure with a simple heuristic. J Chem Inf Comput Sci 43(6):1933–1941. https://doi.org/10.1021/ci034150f
68. Francois B (2018) Bisector tree implementation in OCaml. http://github.com/UnixJunkie/bisec-tree. Accessed 2018-12-01
69. Dean J, Ghemawat S (2004) MapReduce: simplified data processing on large clusters. In: Proceedings of the 6th conference on symposium on opearting systems design & implementation, vol 6. OSDI'04. USENIX Association, Berkeley, CA, USA, p 10
70. OCamlPRO (2018) OCaml package manager. http://opam.ocaml.org. Accessed 2018-12-01
71. community, O (2018) OPAM repository. http://github.com/ocaml/opam-repository. Accessed 2018-12-01
72. Louis G (2018) opam-bundle. http://github.com/AltGr/opam-bundle. Accessed 2018-12-01
73. Jérémie D (2018) Universal toplevel for OCaml. http://github.com/ocaml-community/utop. Accessed 2018-12-01
74. Frédéric B, Thomas R (2018) Context sensitive completion for OCaml in Vim and Emacs. http://github.com/ocaml/merlin. Accessed 2018-12-01
75. Microsoft (2018) Visual studio code. https://code.visualstudio.com. Accessed 2018-12-01
76. GitHub (2018) A hackable text editor. http://atom.io. Accessed 2018-12-01
77. OCamlPRO (2018) Indentation tool for OCaml. http://github.com/OCamlPro/ocp-indent. Accessed 2018-12-01
78. Hugo H (2018) Auto-formatter for OCaml code. http://github.com/ocaml-ppx/ocamlformat. Accessed 2018-12-01
79. Bérenger F (2016) Nouveaux Logiciels Pour la Biologie Structurale Computationnelle et la Chémoinformatique. PhD thesis, Paris, CNAM
80. Faulon J-L, Visco DP, Pophale RS (2003) The signature molecular descriptor. 1. Using extended valence sequences in QSAR and QSPR studies. J Chem Inf Comput Sci 43(3):707–720. https://doi.org/10.1021/ci020345w
81. Moreau G, Broto P (1980) The autocorrelation of a topological structure: a new molecular descriptor. Nouv J Chim 4(6):359–360
82. Wand MP (1994) Fast computation of multivariate kernel estimators. J Comput Graph Stat 3(4):433–445. https://doi.org/10.1080/10618600.1994.10474656
83. Francois B (2018) Chemoinformatics tool for ligand-based virtual screening. http://github.com/UnixJunkie/ACPC. Accessed 2018-12-01
84. Huang N, Shoichet BK, Irwin JJ (2006) Benchmarking sets for molecular docking. J Med Chem 49(23):6789–6801
85. Berenger F (2017) UnixJunkie/consent: release for publication. J Cheminform. https://doi.org/10.5281/zenodo.1006728
86. Francois B (2018) Ligand-based virtual screening with consensus queries. http://github.com/UnixJunkie/consent. Accessed 2018-12-01
87. Dolinsky TJ, Nielsen JE, McCammon JA, Baker NA (2004) PDB2PQR: an automated pipeline for the setup of Poisson–Boltzmann electrostatics calculations. Nucleic Acids Res 32:665–667. https://doi.org/10.1093/nar/gkh381
88. Baker NA, Sept D, Joseph S, Holst MJ, McCammon JA (2001) Electrostatics of nanosystems: application to microtubules and the ribosome. Proc Natl Acad Sci 98(18):10037–10041. https://doi.org/10.1073/pnas.181342398
89. Berenger F (2017) UnixJunkie/fragger: release for Publication in F1000R. https://doi.org/10.5281/zenodo.877320
90. Francois B (2018) A protein fragments picker. http://github.com/UnixJunkie/fragger. Accessed 2018-12-01
91. Gouy I (2018) Debian language shootout. http://benchmarksgame-team.pages.debian.net/benchmarksgame. Accessed 2018-12-01
92. Swamidass SJ, Azencott C-A, Daily K, Baldi P (2010) A CROC stronger than ROC: measuring, visualizing and optimizing early retrieval. Bioinformatics 26(10):1348–1356. https://doi.org/10.1093/bioinformatics/btq140
93. Mysinger MM, Carchia M, Irwin JJ, Shoichet BK (2012) Directory of useful decoys, enhanced (DUD-E): better ligands and decoys for better benchmarking. J Med Chem 55(14):6582–6594. https://doi.org/10.1021/jm300687e
94. Danelutto M, Cosmo RD (2012) A minimal disruption skeleton experiment: seamless map and reduce embedding in OCaml. Procedia Comput Sci 9:1837–1846. https://doi.org/10.1016/j.procs.2012.04.202. Proceedings of the International Conference on Computational Science, ICCS 2012
95. Francois B (2018) parany. http://github.com/UnixJunkie/parany. Accessed 2018-12-01
96. Bourgoin M, Chailloux E, Lamotte J-L (2014) Efficient abstractions for GPGPU programming. Int J Parallel Program 42(4):583–600. https://doi.org/10.1007/s10766-013-0261-x
97. Kiselyov O (2014) The design and implementation of BER MetaOCaml. In: Codish M, Sumii E (eds) Functional and logic programming. Springer, Cham, pp 86–102
98. Taha W (2004) A gentle introduction to multi-stage programming. In: Lengauer C, Batory DS, Consel C, Odersky M (eds) Domain-specific program generation, international seminar, Dagstuhl Castle, Germany, March 23–28, 2003. Lecture notes in computer science, vol 3016. Springer, Berlin, Heidelberg, pp 30–50
99. Le Gall S (2018) ounit. http://github.com/gildor478/ounit. Accessed 2018-12-01
100. Cruanes S (2018) qcheck. https://github.com/c-cube/qcheck. Accessed 2018-12-01
101. Weininger D (1988) SMILES, a chemical language and information system. 1. Introduction to methodology and encoding rules. J Chem Inf Comput Sci 28(1):31–36. https://doi.org/10.1021/ci00057a005
102. Daylight Chemical Information Systems Inc. SMARTS. http://www.daylight.com/dayhtml/doc/theory/theory.smarts.html. Accessed 1 Dec 2018
103. Willighagen EL, Mayfield JW, Alvarsson J, Berg A, Carlsson L, Jeliazkova N, Kuhn S, Pluskal T, Rojas-Chertó M, Spjuth O, Torrance G, Evelo CT, Guha R, Steinbeck C (2017) The Chemistry Development Kit (CDK) v2.0: atom typing, depiction, molecular formulas, and substructure searching. J Cheminform 9(1):33. https://doi.org/10.1186/s13321-017-0220-4
104. contributors C (2018) Chemistry development kit. http://cdk.github.io. Accessed 2018-12-01
105. Tosco P, Stiefl N, Landrum G (2014) Bringing the MMFF force field to the rdkit: implementation and validation. J Cheminform 6(1):37. https://doi.org/10.1186/s13321-014-0037-3
106. O'Boyle NM, Banck M, James CA, Morley C, Vandermeersch T, Hutchison GR (2011) Open babel: an open chemical toolbox. J Cheminform 3(1):33. https://doi.org/10.1186/1758-2946-3-33
107. Raymond ES (2004) The art of unix programming. Addison-Wesley Professional, Indianapolis
108. wwwPDB (2008) Protein data bank contents guide: atomic coordinate entry format description version 3.30. wwwPDB, Piscataway, NJ, USA
109. Tripos I (2005) Tripos Mol2 file format. Tripos Inc, St. Louis
110. Pedregosa F, Varoquaux G, Gramfort A, Michel V, Thirion B, Grisel O, Blondel M, Prettenhofer P, Weiss R, Dubourg V, Vanderplas J, Passos A, Cournapeau D, Brucher M, Perrot M, Duchesnay E (2011) Scikit-learn: machine learning in Python. J Mach Learn Res 12:2825–2830
111. Breiman L (2001) Random forests. Mach Learn 45(1):5–32. https://doi.org/10.1023/A:1010933404324
112. Bastian T (2018) ORandForest. http://github.com/tobast/ORandForest. Accessed 2018-12-01
113. Wang L (2017) Owl: a general-purpose numerical library in OCaml. CoRR. arXiv:1707.09616
114. Wang L (2018) Owl-OCaml scientific and engineering computing. http://github.com/owlbarn/owl. Accessed 2018-12-01
115. Abadi M, Barham P, Chen J, Chen Z, Davis A, Dean J, Devin M, Ghemawat S, Irving G, Isard M, Kudlur M, Levenberg J, Monga R, Moore S, Murray DG, Steiner B, Tucker P, Vasudevan V, Warden P, Wicke M, Yu Y, Zheng X (2016) Tensorflow: a system for large-scale machine learning. In: Proceedings of the 12th USENIX conference on operating systems design and implementation. OSDI'16. USENIX Association, Berkeley, CA, USA, pp 265–283
116. Mazare L (2018) tensorflow-ocaml. http://github.com/LaurentMazare/tensorflow-ocaml. Accessed 2018-12-01

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 svmpath. 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) cpm—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):7. 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. Pettersen 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