Modularité : Notes de cours

LU3IN032 : Programmation comparée

Basile Pesin

12 mars 2021

Table des matières

Résumé

De tout temps, l'humanité a cherché à écrire du code modulaire et réutilisable.

1 Structurer le code : modules et interfaces

Séparer le code en module permet de structurer le code. En plus de rendre le code plus facilement compréhensible et manipulable pour le programmeur, cela permet aussi de permettre la définition de bibliothèques logicielles, comme des collections naturelles de modules, et donc permet la réutilisation de sections de code.

On cherche en général à ce qu'un module remplisse un jeu de fonctionnalités

cohérentes et bien définies. Les modules disposent donc en général d'interfaces, qui peuvent être vues comme une liste de fonctionnalités.

On illustrera cette première section en fournissant en C, Java et OCaml l'interface et l'implémentation (incomplète) d'une structure très simple : un ensemble d'entiers (représenté par une liste triée en ordre croissant, sans doublon).

— L'insertion en place (c'est-à-dire de manière à garder les données triées et à éviter les doublons)
— La recherche (qui exploite l'ordre dans lequel les entiers sont triés)
— L'affichage en ordre croissant

1.1 C : code multi-fichier

Le langage C ne possède pas de système de module formellement défini. Cependant, le langage permet la compilation séparée, c'est-à-dire la construction d'un programme à partir de plusieurs fichiers sources et de bibliothèques.

```c
#ifndef _SORTED_LIST_H
#define _SORTED_LIST_H_

typedef struct _list {
  int data;
  struct _list *next;
} List;

/** Create a new empty set */
List *sorted_empty();

/** Insert an element in the set */
List *sorted_insert(List *s, int data);

/** Check if an element is in the set */
int sorted_mem(const List *s, int data);

/** Print the set in sorted order */
void sorted_print(const List *s);

/** Free the set */
void sorted_free(List *s);

#endif
```

Listing 1 – `list.h`

```c
#include "list.h";

List *sorted_empty() {
  return NULL;
}

List *sorted_insert(List *s, int data) { [...] }

int sorted_mem(const List *s, int data) { [...] }

void sorted_print(const List *s) { [...] }

void sorted_free(List *s) { [...] }
```

Listing 2 – `list.c`

Tout fichier ayant besoin d'utiliser notre "module" peut `#include ‘‘list.h’’` . Par exemple un programme `main.c` utilisant les listes peut etre compilé avec :

```
gcc -c -o list.o list.c
gcc -c -o main.o main.c
gcc -o main list.o main.o
```

1.2 Java : POO à la rescousse

Java est un langage purement orienté-objet [1]. Ce paradigme offre une approche naturellement modulaire. On peut en effet voir chaque classe comme un petit module, exposant un certain nombre de fonctionnalités via ses attributs publics (qui forment son interface).

```
public class SortedList {
    private ArrayList<int> data;

    public SortedList() { data = new ArrayList<>(); }

    public void insert(int t) { [...] }

    public boolean mem(int t) { [...] }

    @Override
    public String toString() { [...] }
}
```

Listing 3 – `SortedList.java`

1.3 Le système de modules d'OCaml

Si Java présente une solution typiquement orientée-objet au problème de la modularité, le langage OCaml, en plus de disposer d'objets, propose aussi un système de modules complet [2], [3].

La façon la plus simple de définir un module en OCaml est, comme pour le C, par fichier. On définit un fichier `.mli` (4) qui contient l'interface du module, ainsi qu'un fichier `.mli` (5) qui contient son implémentation. Seules les valeurs décrites dans l'interface pourront être vues et utilisées par un utilisateur du module. Le reste est "privé". La vérification de types d'OCaml vérifie que les valeurs données dans l'implémentation correspondent bien à celles déclarées dans l'interface. Si une interface explicite n'est pas donnée, la signature implicite du module est calculée par l'inférence de types d'OCaml.

Contrairement aux objets java, de tels modules n'encapsulent pas de données. On peut voir ces modules comme des classes ne contenant que des champs statiques.

```
type t

val empty : t
val insert : int -> t -> t
val mem : int -> t -> bool
val to_string : t -> string
```

Listing 4 – `sortedList.mli`

```
type t = int list

let empty = []

let rec insert d l =
  match l with
  | [] -> [d]
  | hd::tl when (d < hd) -> d::hd::tl
  | hd::tl when (d = hd) -> hd::tl
```

```
  | hd::tl -> hd::(insert d tl)

let rec mem d l =
  match l with
  | [] -> false
  | hd::_ when (d = hd) -> true
  | hd::_ when (d < hd) -> false
  | _::tl -> mem d tl

let to_string l =
  String.concat "␣" (List.map string_of_int l)
```

Listing 5 – `sortedList.ml`

On peut également définir plusieurs modules par fichiers, avec signatures explicites ou non.

```
module type Sorted = sig
  type t
  val empty : t
  val insert : int -> t -> t
  val mem : int -> t -> bool
  val to_string : t -> string
end
```

Listing 6 – Signature de module

```
module SortedList : Sorted = struct
  type t = int list
  let empty = []
  let rec insert d l = [...]
  let rec mem d l = [...]
  let to_string l = [...]
end
```

1.4 Unités de compilation et unités logiques

Dans les exemples précédants, on a vu que les compilateurs permettaient de décomposer un programme un plusieurs unités de compilations (fichiers). `gcc`, `ocamlc` et `javac` permettent tous la compilation séparée :

- `.c` + `.h` -> `.o`
- `.java` -> `.class`
- `.ml` + `.mli` -> `.cmo` + `.cmi`

C'est un aspect technique des compilateurs, qui ne fait pas partie du "langage" à proprement parler. On a vu qu'en C, cette structuration en unités de compilations est en fait la seule possible. En revanche, en Java comme en OCaml, on ajoute une structuration plus "logique", qui fait partie du langage.

De manière astucieuce, les compilateurs font correspondre unités logiques et unités de compilation. En OCaml, le fichier `sortedList.ml` définit un module qui peut être ouvert avec `open SortedList`. De même, en Java, une classe publique `SortedList` doit être définie dans un fichier `SortedList.java` (qui sera donc compilé en `SortedList.class`). Java va même plus loin, puisque le chemin d'un fichier (par exemple `pcomp/sorted/SortedList.java`) correspond exactement au package (`pcomp.sorted.SortedList.java`).

2 Une interface pour plusieurs implémentations ?

L'un des intérêts d'une interface est de pouvoir "cacher" certains détails d'implémentation sur lesquels l'utilisateur du module ne devrait pas s'appuyer. Cela signifie que l'utilisation d'une in-

terface doit être totalement indépendante des choix d'implémentation. En particulier, il est tout à fait possible de donner plusieurs implémentations différentes d'une interface, fonctionnellement équivalentes (c'est-à-dire qu'elles produisent les mêmes effets du point de vue de l'utilisateur), mais pas équivalentes en terme de performances par exemple.

On va voir dans cette partie comment C, Java et OCaml nous permette de changer de manière transparente d'implémentation. On illustrera ce procédé par la définition d'une nouvelle implémentation par arbres binaires pour notre ensemble d'entiers. Celle-ci devrait en effet permettre une recherche plus rapide que l'implémentation par liste.

2.1 C : bricolages de compilation

Comme on l'a vu, la modularité en C ne peut être accomplit qu'en assemblant plusieurs fichiers, et n'est pas réellement supportée au niveau du langage. De manière prévisible, cela signifie que pour interchanger les implémentations, il faut fournir différents fichiers objets (.o) donnant les définitions d'une même interface (.h) lors de l'édition des liens.

Par exemple, si on avait deux fichiers `list.c` et `tree.c` remplissant une interface commune `sorted.h`, on pourrait utiliser les commandes `gcc -o main list.o main.o` ou `gcc -o main tree.o main.o` pour produire un exécutable.

Comment définir cette interface ? La difficulté est que la structure de données manipulée par `list.c` n'est pas la mème que celle manipulée par `tree.c`, ce qui signifie que les signatures fournies dans l'interface ne peuvent pas dépendre de ces structures/types. On est donc obligé de donner aux fonctions de l'interface des types génériques, en utilisant le type `void *` pour représenter une structure abstraite (voir 7).

```
/** Create a new empty set */
void *sorted_empty();

/** Insert an element in the set */
void *sorted_insert(void *s, int data);

/** Check if an element is in the set */
int sorted_mem(const void *s, int data);

/** Print the set in sorted order */
void sorted_print(const void *s);

/** Free the set */
void sorted_free(void *s);
```

Listing 7 – `sorted.h`

```
#include "sorted.h"

typedef struct _list {
  int data;
  struct _list *next;
} List;

void *sorted_empty() { return NULL; }

void *sorted_insert(void *s, int data) {
  List *l = (List *) s;
  [...]
}

int sorted_mem(const void *s, int data) {
  List *l = (List *) s;
  [...]
}
```

```c
void sorted_print(const void *s) {
  List *l = (List *) s;
  [...]
}

void sorted_free(void *s) {
  List *l = (List *) s;
  [...]
}
```

Listing 8 – `list.c`

Les fichiers sources `list.c` et `tree.c` peuvent alors définir les structures concrètes, et implémenter les fonctions. Le problème, comme on le voit en 8, est qu'étant donnée la signature de ces fonctions, on est obligé d'utiliser des opérations de transtypage explicite au début de chaque fonction pour transformer un `void *` en `List *`. Ces opérations peuvent amener a des erreurs de mémoires, en particulier si la donnée passée par l'utilisateur n'est pas réellement une `List *` (ce qui est possible puisque la fonction accepte un `void *`). C'est en réalité la grande faiblesse de cette approche, qui est liée au système de types limité de C. On va voir plus loin comment remédier à ce problème dans nos autres langages.

2.2 Java : Interfaces (avec un grand I)

En Java, on dispose d'un mécanisme bien plus naturel pour définir des interfaces : les `Interface`, qui permettent de lister un ensemble de méthodes qu'une classe doit déclarer. Une classe peut implémenter plusieurs interfaces.

Pour notre exemple, on propose une interface `Sorted` (9). Celle-ci est implémentée par les classes `SortedList` (10) et `SortedTree`.

```java
public interface Sorted {
    public void insert(int t);
    public void mem(int t);
}
```

Listing 9 – `Sorted.java`

```java
public class SortedList implements Sorted {
    private ArrayList<int> data;

    public SortedList() { data = new ArrayList<>(); }

    @Override
    public void insert(int t) { [...] }

    @Override
    public boolean mem(int t) { [...] }

    @Override
    public String toString() { [...] }
}
```

Listing 10 – `SortedList.java`

Les interfaces permettent une forme de "sous-typage" en Java : deux objets issus de classe implémentant `Sorted` peuvent être manipulés de manière équivalente, pour peu qu'on n'en utilise que les champs apparaissant dans l'interface. Par exemple, le code suivant peut être écrit par l'utilisateur :

```java
Sorted[] arr = new Sorted[2];
arr[0] = new SortedList();
```

```
arr[1] = new SortedTree();
arr[0].insert(42); arr[1].insert(42);
[...]
```

De même, l'utilisation des constructeurs `SortedList()` et `SortedTree()` peuvent eux-même être cachés à l'utilisateur (avec le DP "Factory" par exemple). On peut donc totalement abstraire les implémentations de ces modules.

2.3 OCaml

En OCaml, on a bien vu que les signatures de modules étaient séparées déclarées indépendemment de leurs implémentations. Il est donc très simple de déclarer deux modules partageant la même interface. En reprenant l'interface donnée en 6, on peut ajouter l'implémentation par arbres des ensembles comme un nouveau module :

```
module SortedTree : Sorted = struct
  type t = Leaf | Node of (t * int * t)
  let empty = Leaf
  let rec insert d l = [...]
  let rec mem d l = [...]
  let rec to_string l = [...]
end
```

Listing 11 – Module Arbres

Pour l'utilisateur, l'implémentation de ces deux modules est encore une fois totalement cachée par l'interface `Sorted`.

2.4 Une implémentation pour plusieurs interfaces

Si l'interface permet de cacher les détails d'implémentations à l'utilisateur, elle peut également être utilisées pour cacher certaines fonctionnalités.

Par exemple, considérons un objet `PlayerCharacter`, encapsulant les données relatives au personnage d'un jeu en ligne. Supposons que cet objet soit partagé sur le réseau, et donc accessible par le joueur le contrôlant (par le biais d'un cliant de jeu), et par le serveur de jeu. Cet objet n'offre alors pas alors la même interface au joueur et au serveur : le joueur peut contrôler les déplacements du personnage, lui faire effectuer un certain nombres d'actions, etc. Le serveur lui, peut repercuter sur le joueur les effets de son environnement : prise de dégats, etc... Les interfaces de l'objet reflète le fait que les deux parties ont des droits différents sur l'objet.

3 Modules paramétrés

Jusqu'ici, on a décrit un ensemble capables de stocker des entiers, et seulement des entiers. On aimerait pouvoir faire de même pour tout type de donnée :

— Supportant une comparaison formant un ordre total sur les données

— Affichable

On pourrait bien sur réécrire ce code pour chaque type de donnée qu'on voudrait supporter (entiers, flottants, chaînes de caractères, ...) mais on ne pourrait alors pas vraiment dire que ce module est "ré-utilisable". On va montrer ici comment paramétrer notre module pour qu'il supporte n'importe type de données répondant à ces contraintes. On présente les solutions de Java et OCaml, puisque la solution C revient une fois de plus à utiliser des `void *`, ce qui est peu satisfaisant.

3.1 Génériques Java

Java présente un mécanisme de "génériques" permettant de paramétrer des classes, ou interfaces, par des types. Par exemple, en 12, on paramètre l'interface `Sorted` par un type `T`, qui peut ensuite être utilisé dans les signatures des méthodes `insert` et `mem`.

```
public interface Sorted<T> {
    public void insert(T t);
    public boolean mem(T t);
}
```

Listing 12 – Sorted générique, en Java

Comment alors déclarer les classes implémentant cette interface ? Comme on l'a décrit plus haut, le type `T` doit répondre à certaines contraintes : il doit être affichable, et comparable. En Java, toutes les classes disposent d'une méthode `toString` (qui peut être redéfinies), donc il suffit de s'intéresser au problème de comparaison. La bibliothèque standard de Java défini l'interface `Comparable`, qui expose la méthode `int compareTo(T o)` permettant de compararer l'objet courant (`this`) à un autre (`o`). L'interface `Comparable` est héritée par la plupart des classes encapsulant des types primitifs de Java (`Integer`, `Double`, `String`, ...). Les signatures de nos classes seront donc :

```
public class SortedList<T extends Comparable> implements Sorted<T>
public class SortedTree<T extends Comparable> implements Sorted<T>
```

Comme auparavant, elles sont interchangeable du point de vue de l'utilisateur (tant qu'utilisées comme `Sorted<T>`, et peuvent être instanciées pour n'importe quel type héritant de `Comparable`.

3.2 Foncteurs OCaml

En OCaml, l'approche est un peu différente. En effet, au lieu de générique, OCaml propose types paramétrés et polymorphisme. Par exemple, le type des arbres binaires contenant des éléments de type `'a` peut être défini comme :

```
type 'a tree = Leaf | Node of (tree * 'a * tree)
```

A première vue, il semble qu'on peut utiliser cette fonctionnalité pour implémenter nos structures de données génériques. Dans 13, on fourni une signature `Sorted` : les données stockées sont de type `data`, et la structure à la type `t` (dont l'implémentation dépendra du type `data`).

```
module type Sorted = sig
  type data
  type t

  val empty : t
  val insert : data -> t -> t
  val mem : data -> t -> bool
  val to_string : t -> string
end
```

Listing 13 – Sorted générique, en OCaml

Cependant, comment savoir si ce `data` est comparable et affichable ? La solution est de fournir de demander à l'utilisateur de fournir des fonctions `compare : data -> data -> int` permettant de comparer deux données, et `to_string : data -> string` permettant d'afficher une donnée. Comme ces fonctionnalités sont indépendantes, on va les placer chacune dans un module : voir 14 et 15. Ces deux interfaces peuvent aisément être instanciées pour les types de bases, en utilisant des fonctions de la bibliothèque standard OCaml.

```
module type Comparable = sig
  type data
  val compare : data -> data -> int
end
```

Listing 14 – Type comparable

```
module type Printable = sig
  type data
  val to_string : data -> string
end
```

Listing 15 – Type affichable

On peut maintenant écrire des foncteurs pour nos deux implémentations : un foncteur prend des modules en paramètre et construit un nouveau module, qui peut utiliser les fonctions définies dans les modules passés.

On présente en 16 le foncteur `SortedList`, qui prend en paramètre des modules `Comparable` et `Printable`, et retourne un `Sorted`. On contraint les types `data` des deux modules d'entrées à être égaux au type `data` du module construit. Par ailleurs, il faut aussi exposer dans l'interface cette égalité. Dans le cas contraire le type `data` du module `Sorted` serait abstraite, et l'utilisateur ne pourrait donc pas passer de valeur aux fonctions `insert` et `mem`.

```
module SortedList
    (C : Comparable)
    (P : Printable with type data:=C.data)
  : Sorted with type data = C.data = struct
  type data = C.data
  type t = data list

  let empty = []

  let rec insert d l =
    match l with
    | [] -> [d]
    | hd::tl when (C.compare d hd < 0) -> d::hd::tl
    | hd::tl when (C.compare d hd = 0) -> hd::tl
    | hd::tl -> hd::(insert d tl)

  let rec mem d l =
    match l with
    | [] -> false
    | hd::_ when (C.compare d hd = 0) -> true
    | hd::_ when (C.compare d hd < 0) -> false
    | _::tl -> mem d tl

  let to_string l =
    String.concat "␣" (List.map P.to_string l)
end
```

Listing 16 – Liste triée générique, en OCaml

On présente en 17 un exemple d'utilisation de ce foncteur (et `SortedTree`) avec des modules répondant aux interfaces `Printable` et `Comparable` pour le type `int`.

```
module IntCompomp = struct
  type data = int
  let compare = Int.compare
end

module IntPrint = struct
  type data = int
  let to_string = string_of_int
```

```
end

module SListInt = SortedList(IntComp)(IntPrint)
module STreeInt = SortedTree(IntComp)(IntPrint)
```

Listing 17 – Exemple d'utilisation de foncteurs

4 Inclusion, extension, redéfinition

Une propriété intéressante d'un système de modules est de permettre à l'utilisateur d'étendre ou de modifier des modules existants. Par exemple, un utilisateur pourrait vouloir définir une nouvelle version de l'arbre binaire de recherche, muni d'une méthode permettant de ré-équilibrer l'arbre. Par ailleurs, il pourrait aussi vouloir redéfinir la méthode d'insertion, de manière à ce que l'arbre soit rééquilibré automatiquement après chaque insertion.

4.1 Java : héritage

Java propose la notion d'héritage de classes : il est possible de définir des classes "héritant" des fonctionnalités d'une classe existante. Ces classes "filles" peuvent accéder à tous les attributs marqués `public` ou `protected` de la classe "mère" héritée.

```
public class BalancingSortedTree<T extends Comparable> extends SortedTree<T> {
    public void rebalance() { [...] }

    @Override
    public void insert(T t) {
        super.insert(t);
        rebalance();
    }
}
```

Listing 18 – Arbre binaire de recherche auto-équilibré, en Java

En 18, on montre à quoi la classe Java étendant l'arbre binaire de recherche ressemble. On note que la méthode `rebalance` accède directement aux champs de "données" de l'arbre, qui doivent donc être marquées `protected` plutôt que `private`. La nouvelle fonction d'insertion utilise elle directement la fonction d'insertion de la classe mère, à laquelle elle accède via le mot clé `super`.

4.2 OCaml : inclusion de modules

Comme Java, OCaml permet de définir de nouveaux modules utilisant les "héritant" de modules existants. On appelle ce mécanisme "inclusion" de modules.

```
module BalancingSortedTree
    (C : Comparable)
    (P : Printable with type data:=C.data)
  : Sorted with type data = C.data = struct
    include SortedTree(C)(P)

    let rebalance t = [...]

    let insert d t =
      rebalance (insert d t)
end
```

Listing 19 – Arbre binaire de recherche auto-équilibré, en OCaml

En 19, on voit comment utiliser cette fonctionnalité pour implémenter notre extension de l'arbre binaire de recherche. Le mot clé `include` permet de spécifier le (ou les modules) étendus. Contrairement à Java, on peut en effet inclure plusieurs modules à la fois.

On note que dans la nouvelle méthode `insert`, on appelle directement `insert` : contrairement à Java, il n'y a pas de mot clé signifiant que la fonction doit être récupérée dans un des modules inclus. En fait, ici c'est parce qu'`insert` n'est pas déclarée récursive que le compilateur déduit qu'elle est définit "plus haut", c'est à dire dans le module `SortedTree` inclus.

On note également que dans cette implémentation, la fonction `rebalance` n'est pas visible à l'extérieur du module, puisque le module est contraint par la signature `Sorted`.

5 Bonus : Fonctionnalités avancées, tout le monde s'influence

5.1 Modules de première classe

En Java, on peut écrire :

```
Sorted<Integer> s;
if(b) { s = new SortedList<>(); }
else { s = new SortedTree<>(); }
```

Autrement dit, le résultat d'un calcul permet d'influencer le choix de l'implémentation. C'est utile, par exemple si on veut pouvoir choisir une implémentation en passant un paramètre à l'exécution du programme.

En OCaml, on ne peut pas à priori faire cela, car la déclaration et la construction de modules (par application de foncteurs) ne "vit" pas dans le même langage que les expressions classiques : OCaml est en effet stratifié en langage "noyau" d'expressions, et langage de modules. Le second contient le premier, mais pas le contraire.

Heureusement, il existe une solution en OCaml : les modules de première classe permettent de manipuler des modules comme valeurs de première classe. On peut par exemple, avec les déclarations de `SListInt` et `STreeInt` données plus haut, écrire le code suivant :

```
let (module S : Sorted with type data = int) =
  if b then (module SListInt)
  else (module STreeInt)
```

La manipulation de modules n'est donc pas plus limitée que l'instantiation de classes, heureusement.

5.2 Packages et Modules Java

Jusqu'ici, on a principalement parlé de classes en Java. Mais il se trouve que depuis sa 9eme version, Java dispose également d'un système de modules ([4]). Ce système est en fait très différent de la modularité qu'on a vu jusqu'ici, puisqu'il agit a beaucoup plus gros grain.

En Java les classes sont regroupées dans des packages, qui correspondent en fait à la hiérarchie des fichiers sources. Par exemple, pour les classes données plus tôt, on pourrait proposer la hiérarchie suivante :

```
src/
  pcomp/
    interfaces/
      Sorted.java
    implems/
      SortedTree.java
      SortedList.java
      BalancingSortedTree.java
```

Dans cette configuration, on ajouterait au début des fichiers `Sorted.java` la ligne `package pcomp.sorted.interfaces;` et au début de `SortedList.java`, `SortedTree.java` et `BalancingSortedTree.java` la ligne `package pcomp.sorted.implems;`.

Ce système de packages permet de qualifier les classes, et donc de gérer l'utilisation d'une classe dans une autre : les classes dans le même package ont accès "par défaut" les unes aux autres. En revanche, il faut manuellement "importer" une classe déclarée dans un autre package : par exemple, dans `SortedTree.java`, il faut utiliser `import pcomp.interfaces.Sorted;`.

Le systèmes de modules construit au dessus du système de packages : il permet de définir un module comme un ensemble de packages. Il permet de mieux structurer la distribution de modules. En particulier, il permet de définir des relations de dépendances entre modules, où un module utilise le code d'un autre. Cela permet d'éliminer à la compilation des erreurs (bibliothèque non installée, etc...) qui n'apparaissaient auparavant qu'à l'exécution.

Ci dessous, on montre comment définir un module pour notre petite bibliothèque. Celui ci exporte le contenu des packages `interfaces` et `implems`, et indique qu'il utilise le package `java.utils` (car on utilise la classe `ArrayList`).

```
module pcomp.sorted {
    requires java.utils;
    exports pcomp.sorted.interfaces;
    exports pcomp.sorted.implems;
}
```

Listing 20 – Définition de module

On peut ensuite compiler l'ensemble des classes, en ajoutant le fichier `module-info.java` dans le chemin de compilation :

```
javac -d classes/sorted \
        src/pcomp.sorted/module-info.java \
        src/pcomp.sorted/pcomp/sorted/interfaces/*.java \
        src/pcomp.sorted/pcomp/sorted/implems/*.java
jar -c -M -f sorted.jar -C classes/sorted .
```

L'archive `sorted.jar` fourni doit alors être incluse avec l'option `--module-path` quand on compile ou exécute un autre module dépendant du module `pcomp.sorted`.

Références

[1] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, Daniel Smith, and Gavin Bierman. *The Java Language Specification.* Oracle, 2020.

[2] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The OCaml system : Documentation and user's manual.* Inria, 4.11 edition, August 2020.

[3] Yaron Minsky, Anil Madhavapeddy, and Jason Hickey. Real World OCaml : Functional programming for the masses - files, modules, and programs. 2020.

[4] Mark Reinhold. The State of the Module System. 2020.