Software Verification

2023-2024

Rapide tuto sur OCaml

Vincent Penelle

Points abordés

- Rappels OCaml

OCaml est le langage que nous utiliserons durant ce cours. Le but de ce document est plus de servir d'aide-mémoire pratique qu'autre chose : on va faire le point dessus ensemble. N'y passez ensuite pas trop de temps. Si vous n'êtes pas familiers de OCaml, vous pouvez trouver des tutoriaux ici[1], ou, pour des interrogation précises, vous tourner vers la documentation[2]. On va résumer ici quelques notions qui nous serons utiles avec quelques courts exercices qui abordent ces notions et syntaxes.

Télécharger l'archive du TP[3], et ouvrez le fichier `exo1.ml`. Il contient ce qui est discuté ici, ainsi que les prototypes des petits exercices.

Avant de commencer, vous devez taper les commandes suivantes dans un terminal depuis lequel vous lancerez vs-code (si vous utilisez cet éditeur, sinon, les deux premières commandes sont à taper là où vous compilerez) :

- `export OPAMROOT=/opt/local/opam`
- `eval $(opam env)`
- `code`

Nous vous conseillons d'utiliser l'extension OCaml platform de vs-code si vous utilisez cet éditeur.

Exercice 1: Utilisation et Compilation: Dans le cadre du cours, nous implémenterons un logiciel où nous avons automatisé la compilation via un Makefile. Cependant, si vous souhaitez tester des fonctions/programmes séparément, vous avez plusieurs manières de le faire. La plus rapide (mais moins durable) et de lancer le *toplevel*. Vous pouvez le faire avec la commande `ocaml`, mais il n'est pas très commode. Un toplevel bien plus évolué existe : `utop` (il est installé au CREMI). Il est cependant possible d'ouvrir un fichier .ml dans le toplevel.

L'autre option est de créer un fichier `MonFichier.ml` et de le compiler. La compilation avec le compilateur de base, `ocaml` est similaire à gcc : pour des projets très simples,

[1] `http://OCaml.org/learn/`
[2] `http://caml.inria.fr/pub/docs/manual-OCaml/`
[3] `TP-SMT.tar.gz`

il suffit, mais dès qu'on a un projet un peu gros avec des bibliothèques à inclure et des dépendances, ce n'est pas commode.

Il existe heureusement des builders automatiques pour OCaml. Le plus ancien (et qui commence à être déprécié) est `ocamlbuild`. Un plus récent, et que nous utiliserons ici, se nomme `dune`. Nous avons caché le procédé sous un Makefile, vous pourrez donc compiler l'exercice avec `make exo1.native` (version compilée en natif) ou `make exo1.byte` (version compilée en bytecode). Vous pouvez directement utiliser dune avec la commande `dune build`, mais dans ce cas, les fichiers compilés le seront dans le dossier `_build/default`.

Dune permet également de charger simplement les bibliothèques d'un projet sur lequel vous travaillez dans utop assez simplement. Pour cela, il suffit de taper `#use_output "dune top";;` dans une session utop, et vous aurez alors accès aux modules présents dans les bibliothèques du projet sur lequel vous travaillez. Pour le projet courant, dune est configuré pour que ce soit le cas des modules concernant Keen (donc pas ce tuto). Cela peut vous permettre de tester rapidement des fonctionnalités que vous développez dans leur contexte sans avoir à écrire un main.

Dune est un système de build complet très configurable, et de ce fait pas si simple à prendre en main. Le but de ce tutoriel n'est pas de vous expliquer comment faire cela, nous vous renvoyons à sa documentation si vous souhaitez vous pencher dessus. Dans le cadre de ce cours, nous vous fournirons les projets configurés avec dune de manière à ce que son utilisation soit transparente pour vous.

Exercice 2: Langage Fonctionnel: Nous nous servirons du paradigme fonctionnel du langage (on peut techniquement faire de l'impératif, mais c'est moins élégant – et surtout moins adapté à notre projet). Un fichier est exécuté de haut en bas (comme en C, il peut comporter plusieurs fichiers : si une fonction d'un autre fichier est appelée, dune ira le compiler – c'est en partie à cause de ça qu'à la main c'est compliqué).

Le fichier est une suite de requêtes qui vont modifier la mémoire du programme. Une requête est de la forme `let name = expr`, où name est une chaîne de caractères, et expr une expression du langage. Par exemple, `let toto = 4+2` mettra 6 à l'adresse toto de la mémoire (c'est assez mal dit, mais c'est l'idée). On peut aussi évaluer une expression sans la retenir en écrivant `let _ = expr`, et préciser qu'on attend une action avec `let () = expr` (cf plus bas).

Tous les éléments manipulés par le langage sont des fonctions (une constante étant simplement une fonction sans argument). On peut donc définir une fonction avec la requête `let`, en nommant les arguments. Par exemple, `let f x = x + 1` définira la fonction f qui prend un entier x et renvoie x+1. Il n'est pas nécessaire de préciser les types (le compilateur les infère et signale les inconsistances). La requête `let ... in` définit une fonction localement. Cela est en général utile dans d'autres définitions (et tient lieu de ≪terminateur d'expression≫). Par exemple, `let f x = let succ x = x+1 in if(x < 0) then 0 else succ x`.

Il y a également des expressions de type `unit` qui sont des actions qui sont exécutées lorsqu'elles sont définies, par exemple, l'affichage avec `Printf.printf`. Dans ce cas, il n'est pas nécessaire de leur donner un nom et on peut simplement écrire `let _ = Printf.printf "Toto"`. On peut bien évidemment en utiliser dans d'autres expressions (avec `let ... in`), auquel cas l'action sera effectuée à chaque fois que l'expression est évaluée. On peut alternativement utiliser le `;` pour séquencer deux expression : `A ; B` sera la même chose que `let _ = A in B`.

Dans le toplevel, il est nécessaire d'utiliser `;;`, pour lancer l'évaluation d'un bloc de code. Dans les fichiers compilés, c'est inutile (mais ils n'empêchent pas la compilation).

Comme dit avant, tous les objets manipulés par le programme sont des fonctions. Il est donc évidemment possible d'avoir des fonctions comme argument d'autres fonctions. Par exemple `let g f x = (f x) + 1` définit g qui prend en argument une fonction f de `'a -> int` et un `'a` x et renvoie le résultat de f x auquel est ajouté 1. Ici, `'a` désigne un type quelconque : le langage supporte les fonctions polymorphes, et donc cette fonction pourra être utilisée sur n'importe qu'elle fonction dont le type de retour est un int et qui dispose d'un seul argument. Par exemple `g int_of_string "25"` est l'int 26.

Autre exemple `let disjunct f g b = if(b) then f else g` est une fonction bien définie (f et g doivent avoir le même type et peuvent accepter des arguments). Enfin, on peut utiliser la notation `fun` pour définir des fonctions anonymes (ou autres utilisations) comme suit : `let res = (disjunct (fun x -> if(x < 0) then 0 else x+1) (fun x -> x-1) true) 3` est un int bien défini (ici, c'est 3). On peut évidemment utiliser la syntaxe `let f = fun x -> x+1`.

Point de syntaxe important : en OCaml, l'égalité *structurelle* se note `=` et pas `==` comme dans la plupart des langages. `==` existe, mais a un autre sens (égalité physique). De même, c'est l'inégalité structurelle est `<>` et `!=` l'inégalité physique. En général, ce que vous voulez, c'est l'égalité structurelle. Pour vous en convaincre, vous pourrez remarquer que `let a = 1.1 in a = 1.1` est vrai alors que `let a = 1.1 in a == 1.1` est faux.

Autre point : attention au parenthésage : l'associativité est à gauche, donc un terme du type `f x+1` sera interprété comme `(f x) + 1`. Il est préférable de mettre des parenthèses pour désambiguïser les expressions. À noter cependant que le passage d'arguments à une fonction se fait sans parenthèses (ce qu'on écrirait `f(1,2)` en C s'écrit bien `f 1 2`).

Exercice: Définir une fonction `compo` qui prend en argument deux fonctions (quelconques) et renvoie leur composition (attention, ici, on ne peut pas ne pas mettre l'argument).

Exercice 3: Récursion et récursion terminale: Un concept versatile et important en fonctionnel, c'est la récursion. Quand on définit une fonction récursive, il faut cependant l'indiquer au compilateur avec le mot clé rec comme suit : `let rec toto x = if (x < 0) then 0 else 1 + (toto (x-1))`. Sans rec, le compilateur refusera de compiler (sauf si toto est déjà défini, puisque la redéfinition est autorisée, et c'est d'ailleurs pour cette raison qu'il faut indiquer rec ou non).

Cependant, le défaut de la récursion est qu'il faut utiliser la pile d'appel et que celle-ci n'est pas illimitée. Par exemple, si vous appelez `toto 10000000`, votre programme fera un Stack Overflow, alors qu'avec une boucle, il n'y aurait pas de problème. Fort heureusement, OCaml est optimisé pour éviter d'utiliser la pile d'appel dans le cas d'une *récursion terminale*, c'est-à-dire une fonction récursive dont l'appel récursif est la *dernière* instruction. Toutes les fonctions ne sont pas ré-écrivables simplement de manière récursive terminale, mais beaucoup le sont (il existe des techniques pour trouver une version récursive terminale d'une fonction, celle présentée dans ce qui suit est simple mais pas toujours applicable). Par exemple, on peut réécrire toto de la manière suivante : `let toto x = let rec aux x acc = if(x < 0) then acc else aux (x-1) (acc+1) in aux x 0` On notera que c'est aux qui est maintenant récursive et plus toto. Cette technique d'utiliser une accumulateur pour convertir la fonction en fonction récursive terminale est généralement une bonne méthode.

Exercice: Définir la fonction $x \to 2^x$ de manière récursive terminale.

Exercice plus dur: Définir la fonction de Fibonacci de manière récursive terminale.

Exercice 4: Les types et le pattern matching En OCaml, les types de base sont les mêmes qu'en C (int, float, bool, char, etc). Il existe également des types complexes prédéfinis utilisables sans les définir, par exemple les tuples ($(2,'c')$ est de type int*char),

où les fonctions ($int \to int * char \to char$ est le type des fonctions prenant un int et unt int*char comme argument et renvoyant un char). Il est tout à fait possible de les utiliser directement (et le compilateur y arrivera très bien).

On peut nommer des types avec la requête `type`. Par exemple `type toto = int*char`. Mais la plus grande utilité de cette requête va être de pouvoir avoir des ≪types collection≫, qui seront définis par des constructeurs. On peut les définir comme suit : `type toto = None | One of int | Two of int*char` définit un type qui peut contenir soit rien, soit un entier, soit une paire int*char. On peut également définir des types récursifs (exemple `type tree = Leaf of int | Node of tree*tree`). Note, les constructeurs doivent commencer par une majuscule et les types par une minuscule.

Mais ces types seraient peu utilisables sans l'une des fonctionnalités les plus puissantes d'OCaml, le pattern matching: il permet de décapsuler le contenu d'un type et de renvoyer des valeurs différentes en fonction du cas dans lequel on est (tant que toutes les valeurs ont le même type, bien sûr). Par exemple, la somme des éléments d'un arbre défini plus haut peut être définie comme suit: `let rec sum_of_tree t = match t with Leaf a -> a | Node (t1,t2) -> (sum_of_tree t1) + (sum_of_tree t2)`.

On peut mettre des motifs plus complexes dans le pattern matching, ne pas nommer certains éléments si on n'en a pas besoin (avec `_`) et faire un cas ≪pour tout le reste≫ (avec `_` également). Par exemple: `let rec silly t = match t with Node(Leaf _,_) -> 1 | | Node(_,t1) -> 1 + silly t1 | _ -> failwith "arg"`. On notera qu'ici mon cas par défaut est différent : il renvoie une exception qui terminera le programme si elle n'est pas rattrapée (cela peut être utile dans des cas qui ne sont pas censés arriver et où au lieu de mettre une valeur, on souhaite simplement que le programme plante). L'inférence de type n'aura aucun problème avec ça. Il est toujours utile de faire cela car les pattern-matching sont censé être exhaustif (couvrir tous les cas).

Enfin, on peut mentionner le mot-clé when qui permet de restreindre le pattern à certaines valeurs seulement : `let toto a = match a with Some a when a < 0 -> 0 | Some a -> a | None -> 0`. À noter que dans les deux cas précédents, il n'y a pas d'ambiguïté : le pattern matching est parcouru de haut en bas et s'arrête dès le premier match. L'ordre est donc important. Par ailleurs, les constructeurs None et Some sont les constructeurs du type option qui est un type générique permettant à une fonction de ne pas renvoyer de valeurs dans certains cas (None). Cela permet d'éviter d'utiliser des exceptions dans le flux normal du programme (et c'est très utile).

Exercice : Définissez un type représentant une direction de déplacement (selon les 4 points cardinaux) et une vitesse (ainsi qu'une valeur disant de rester sur place). Définissez aussi une fonction appliquant un objet de type précédent à une position, mais qui ne fait rien si la vitesse est négative.

Exercice 5: Les listes: On utilisera beaucoup les listes dans notre projet. Vous pouvez regarder la page du manuel `https://ocaml.org/api/List.html` pour plus d'informations. On va simplement résumer les bases ici.

Une liste est une collection d'éléments linéairement ordonnée. On a accès à la tête de liste immédiatement. On peut facilement ajouter un élément en début de liste (temps constant), moins en queue (temps linéaire). La liste vide est nommée `[]`. `a::l` représente la liste `l` précédée de `a`. Par exemple `1::2::[4;5] = [1;2;4;5]`. On a les fonction `hd` et `tl` définie ainsi: `hd (a::l) = a` et `tl (a::l)` = l. Il est évidemment possible de faire du pattern matching sur les listes.

Cependant, la grande utilité du module Liste repose dans ses fonctions pour des opérations sur des listes. On va présenter les trois qui sont les plus versatiles et nous serons les plus utiles :

- `map` est une fonction qui va appliquer une fonction à tous les éléments d'une liste. Son type est `('a -> 'b) -> 'a list -> 'b list`. Par exemple `List.map (fun x -> (int_of_string x) + 1) ["12";"0";"-2"]` est la liste `[13;0;-1]`.

- `iter` est une fonction qui va appliquer récursivement à tous les éléments d'une liste (de gauche à droite) une fonction de type unit. Elle est de type `('a -> unit) -> 'a list -> unit`. Par exemple, `List.iter (Printf.printf "%d ++ ") [1;2;3]` affichera "1 ++ 2 ++ 3 ++ ".

- `fold_left` est très similaire à la précédente, sauf qu'elle s'occupe de fonctions qui ont une valeur de retour qu'elle peuvent reprendre en argument. Son type est `('a -> 'b -> 'a) -> 'a -> 'b list -> 'a list`. `List.fold_left f a [b1;b2;b3]` est la même chose que `f (f (f a b1) b2) b3`. a est la valeur d'initialisation, et la valeur de retour peut être vu comme un accumulateur. C'est une manière pratique d'itérer sur une liste sans utiliser de boucle. Par exemple `List.fold_left (fun x y -> x + y) 1 [2;3;4]` renverra 10.

À noter que les trois fonctions précédentes sont récursives terminales (et donc ne peuvent générer de stack overflow). Il existe également `List.fold_right` qui applique la fonction dans l'ordre inverse à fold_left, mais elle n'est pas récursive terminale (donc à éviter si possible).

Exercice : Dans l'exercice précédent, appliquez la fonction de déplacement selon une direction et une vitesse à tous les points d'une liste de coordonnées (grâce à map). Déterminez ensuite le point le plus loin de (0,0) (grâce à fold_left).

Exercice 6: Les modules Vous aurez remarqué que dans le paragraphe précédent, j'ai préfixé map, iter, etc, par List. C'est parce que ces fonctions sont déclarées dans le module List. Les modules sont en première approximation (très grossière – et fausse, cf plus loin) l'équivalent des fichiers. La seule convention de nommage est qu'il doivent commencer par une majuscule. Pour appeler une fonction f du module Toto, il y a deux options : soit on fait un `open Toto` plus haut dans le fichier et on pourra ensuite utiliser les noms de Toto comme s'ils étaient définis dans le fichier courant ; soit on appelle simplement `Toto.f`. La deuxième méthode est en général préférée, sauf si un module est très utilisé (et encore) car elle permet d'être certain de ne pas avoir de conflit dans les noms, ou d'avoir des problème de redéfinition. Il est en effet possible de donner le même nom à des fonctions dans des modules différents. Si vous faites `open List`, vous aurez par exemple accès au nom `map`, mais si vous faites plus loin `let map = 3`, alors map désignera bien 3 et plus la fonction de List (et vice-versa si vous le faites dans l'autre sens).

Les modules sont en réalité bien plus riches que cela : rien n'empêche de déclarer des modules à l'intérieur d'autres modules (c'est même courant). Et il existe même des «foncteurs» qui permettent de créer à la volée des modules paramétrés par d'autres modules (le plus utile étant probablement le module Map[4] qui est grossièrement une table de hachage et permet de représenter à peu près tout ce qu'on veut : du moment qu'on a un ordre sur les clés, on peut tout à fait faire des tables indexées par des chaînes de caractères). Nous n'aurons a priori pas vraiment besoin de tout cela dans le cadre de ce cours (dans les fichiers que nous vous fournissons, on joue avec cela, mais ça devrait être assez transparent pour vous). Cependant, si vous voulez utiliser du OCaml pour d'autres programmes, les modules les plus versatiles sont à mon avis List, Map et String.

[4] `https://ocaml.org/api/Map.html`