Ocp-build: un gestionnaire de projets pour OCaml

Fabrice Le Fessant, Thomas Gazagnaire

ocp-build: un gestionnaire de projets pour OCaml

F. Le Fessant[1], T. Gazagnaire [2]

1: INRIA Saclay – Ile-de-France,
F-91893 Orsay
fabrice.le_fessant@inria.fr
2: OCamlPro SAS
thomas.gazagnaire@ocamlpro.com

Résumé

Dans ce papier, nous présentons `ocp-build`, un nouveau gestionnaire de projet pour OCaml[3]. La gestion de projet — description du projet et compilation — a toujours été l'une des faiblesses de l'environnement de programmation d'OCaml. En effet, les règles de compilation d'un projet OCaml peuvent être complexes à mettre en œuvre avec des outils classiques. `ocp-build` est une solution simple et facile d'accès à ce problème, il permet de décrire de façon concise un projet en terme de paquets et de fichiers sources, puis d'utiliser cette description pour compiler efficacement le projet, ou le manipuler avec des outils de refactoring.

1. Introduction

Un gestionnaire de projet permet de décrire les composants d'un projet, tels que ses dépendances, ses fichiers sources et ses options, puis de le compiler efficacement. L'absence d'un tel gestionnaire de projet *reconnu* pour OCaml est une des faiblesses les plus importantes pour la diffusion de ce langage. En effet, la communauté se divise aujourd'hui entre les utilisateurs de `make`, d'`OMake`[2] et d'`ocamlbuild`.

`make` a l'avantage d'être un outil ancien et générique, donc facile d'accès pour les débutants. Des fichiers, tels `OMakeFile`, contitennent des règles de compilation pour `make`, et permettent de compiler des projets simples à partir d'une liste de fichiers. Les deux inconvénients majeur de `make` sont : (i) le manque portabilité sous Windows ; et (ii) la difficulté d'exprimer toute la complexité des règles de compilation des projets OCaml, empêchant par conséquent la compilation parallèle de ces projets. Ce dernier point est important aujourd'hui pour utiliser au maximum l'architecture multicore des machines de développement modernes.

`OMake` est une version plus évoluée de `make`, développée en OCaml, qui améliore sa portabilité en implantant en interne de nombreuses commandes de *shell*. Faute d'alternative crédibles au moment du démarrage de leurs projets, `OMake` est utilisé par la plupart des utilisateurs industriels d'OCaml, tels Jane Street, Lexifi et Citrix. Hélàs, `OMake` n'est plus maintenu, peu diffusé et la sémantique de son langage est complexe. Surtout, `OMake` souffre de problèmes de performance et de stabilité.

`ocamlbuild` est l'outil de compilation diffusé dans la distribution OCaml depuis la version `3.10`. C'est un outil puissant, extensible via un système de plugins, ce qui peut le rendre capable de compiler n'importe quel type de projets et pas seulement ceux ecrits en OCaml. Il peut compiler sans description préalable des projets OCaml simples. Cependant, il présente trois inconvénients majeurs : (i) son langage de plugins demande une connaissance des fonctionnalités avancées d'OCaml (foncteurs, variants polymorphes), ce qui le rend peu abordable pour les débutants ; (ii) la description du projet OCaml, quand elle existe, est disséminées dans de multiples fichiers (`mllib`, `mlpack`, `_tags`), ce qui

la rend peu lisible et difficilement exploitable par des outils externes ; et finalement (iii) c'est un outil trop généraliste pour être réelement efficace ; les règles de compilation d'OCaml qu'il implémente sont incomplètes et ne permettent toujours pas la compilation parallèle.

Depuis peu, quelques projets utilisent `oasis`, un système qui s'inspire librement de `cabal`[1], le gestionnaire de projets d'Haskell. `oasis` permet de décrire des projets simples en OCaml et gènere ensuite un plugin pour `ocamlbuild` (un plugin pour `make` et `OMake` est planifié). C'est une approche intéressante qui essaye de faciliter l'utilisation des outils de compilation existants. Ellt se heurte cependant à deux problèmes : (i) le langage de description de projet, basé sur la description de projets Haskell, n'est pas bien adapté pour exprimer la complexitée des règles de compilation de projets OCaml – pour corriger ce probléme, `oasis` dispose de nombreuses options pour configurer un projet, ce qui peut rendre son utilisation complexe ; et (ii) utilise des outils de compilation externes et hérite des defaults inhérent à cet outil. De plus, comme il utilise l'API de ces outils externes, il doit traduire les les règles de compilation OCaml vers des commandes compréhensibles par ces outils, ce qui constitue une compléxité supplémentaire. L'outil qui permet de *décrire* un projet est aussi celui qui est le mieux placé pour *compiler* ce projet !

Ce rapide survol nous permet de formuler les qualités principales qu'un bon gestionnaire de projet pour OCaml auraient :

Simplicité d'utilisation Le gestionnaire de compilation doit permettre de décrire de la façon la plus simple possible les projets OCaml, en exploitant au maximum sa connaissance du processus de compilation d'OCaml ;

Portabilité Le gestionnaire de compilation doit faire le minimum d'hypothèses sur le système d'exploitation utilisé, le format des noms de fichiers et les outils à sa disposition, afin de fonctionner aussi bien sous Linux, Mac OS X que sous Windows ;

Lisibilité la syntaxe de description des projets doit être facilement lisible, pour permettre d'une part aux nouveaux développeurs de comprendre rapidement l'architecture d'un projet, et d'autre part, à des outils de développement (génération automatique de code, *refactoring*, navigation, documentation) de comprendre les liens entre les paquets, les sources et les fichiers objets générés.

Compilation efficace Le gestionnaire de compilation doit réduire le temps de compilation d'un projet, en exploitant à la fois sa connaissance des règles de recompilation en fonction des dépendances (compilation incrémentale) et aussi l'architecture multicore des machines de développement modernes (compilation parallèle).

Dans cet article, nous présentons `ocp-build`, un nouveau gestionnaire de projet pour OCaml, qui possède l'ensemble de ces qualités. Son langage de description des paquets est spécialisé pour OCaml, ce qui permet de décrire un projet composé de multiples paquets inter-dépendants de façon concise et lisible, tout en ne faisant aucune d'hypothèse limitant sa portabilité. Sa connaissance des subtilités des règles de compilation d'OCaml offre une compilation parallèle et incrémentale du projet, qui permet au développeur d'obtenir un retour rapide pour chacune de ses modifications. Enfin, le format de description des projets est utilisé par les outils de notre environnement de développement pour OCaml, `TypeRex`, pour la navigation dans les sources, le refactoring de code multi-sources, la génération automatique de code et de documentation.

La partie 2 introduit les concepts que nous utilisons dans la suite de l'article et rappelle les règles de compilation OCaml. Nous détaillons ensuite le format de description de projet dans la partie 3, ainsi que le fonctionnement de l'outil de compilation dans la partie 4. La partie 5 est consacrée à la descrition de l'implémentation d'`ocp-build`. Enfin, la partie 6 conclue cet article.

2. Contexte

2.1. Terminologie

Dans la suite, nous utiliserons le terme *paquet* pour décrire un ensemble de *fichiers sources*, qui permettent de générer soit une *bibliothèque*, soit un *programme*. Un *projet* est un ensemble de paquets.

`ocp-build` requiert que les dépendances entre paquets ne forment pas de cycles. Nous utiliserons donc le terme *pré-requis* pour ces dépendances entre paquets, et le terme général *dépendance* dans le cas précis de dépendances entre fichiers. Ces fichiers peuvent être des *fichiers d'origine*, i.e. existant à l'origine avant le démarrage du processus de compilation, des *fichiers finaux*, i.e. que l'on désire obtenir à la fin de la compilation pour les installer, et des *fichiers temporaires*, i.e. qui résultent du processus de compilation, mais ne seront pas utiles après installation. Parmi ces fichiers, nous distinguons les fichiers d'*interface* (`mli` par exemple) des fichiers d'*implémentation* (`ml` par exemple), et les fichiers *texte* (`ml` par exemple) des fichiers *binaires* (`cmo`, `cmi`, etc.).

Un ensemble de paquets est *clos* si tous les pré-requis et tous les fichiers sources de tous les paquets sont présents dans le projet. Une *règle de compilation* décrit une *suite de commandes* à exécuter pour, à partir d'un ensemble de *fichiers sources* pour la règle, générer un *fichier cible* pour la règle, parfois en générant d'autres *fichiers cibles secondaires*. Les règles travaillent parfois sur des *fichiers virtuels*, c'est-à-dire des fichiers qui n'apparaissent jamais sur le disque, mais permettent de décrire des étapes intermédiaires dans le processus de compilation.

2.2. Règles de compilation d'OCaml

Nous rappelons brièvement les règles de compilation d'OCaml :

Calcul des dépendances La commande `ocamldep` permet d'obtenir une approximation des dépendances d'un fichier source. Il s'agit d'une approximation car une dépendance peut être introduite alors qu'elle n'est pas nécessaire. Dans l'exemple suivant :

```
include M
open N
```

`ocamldep` va inférer une dépendance vers les modules `M` et `N`, bien qu'il soit possible que le module `N` soit défini dans le module `M`. `ocamldep` possède deux modes, le mode par défaut, et le *mode abbrégé* (argument `-modules`). Le mode par défaut calcule les dépendances vers les modules, recherche dans les répertoires les sources qui implante ces modules, puis affiche les dépendances au format `make`. Il est important de noter que, dans ce mode, les fichiers textes temporaires (résultant par exemple des commandes `ocamllex`, `menhir` et `ocamlyacc`) doivent avoir été générés au préalable, ce qui peut limiter la parallèlisation de la compilation du projet. Le mode abbrégé se contente lui d'afficher la liste des modules approximés, laissant à l'outil de gestion de projet la tâche de calculer les dépendances vers les fichiers. C'est donc le mode préféré des outils comme `OMake`, `ocamlbuild` et `ocp-build`.

Compilation des interfaces Les compilateur OCaml requièrent qu'un fichier d'interface `mli` soit compilé (vers un fichier binaire `cmi`) avant que son implémentation `ml` le soit. Les deux compilateurs (bytecode et natif) peuvent être utilisés pour compiler une interface.

Compilation des implémentations Un fichier texte d'implémentation peut être compilé à la fois en bytecode, vers un fichier binaire `cmo`, et en code natif (code machine), vers un fichier binaire `cmx`. Si aucun fichier d'interface `mli` n'est disponible, un fichier binaire d'interface `cmi` sera généré. Dans le cas contraire, il est utilisé pour contraindre l'interface de l'implémentation. Il existe aussi une différence importante entre la compilation bytecode et native : les fichiers binaires bytecode `cmo` ne dépendent que des interfaces, alors que les fichiers binaires natifs `cmx`

dépendent les uns des autres, car le compilateur les utilise pour propager les constantes et *inliner* les petites fonctions.

Édition des liens (*link*) L'ensemble des fichiers d'implémentation objets sont finalement rassemblés dans des bibliothèques ou des programmes. Lors de ce link, le compilateur vérifie que tous les modules ont été compilés en utilisant les mêmes fichiers binaires d'interface, et, pour les fichiers binaires natifs, qu'ils ont été compilés en utilisant les mêmes fichiers binaires natifs.

Certaines de ces règles de compilation présentent des propriétés qu'on retrouve rarement dans d'autres langages de programmation, et qui rendent leur parallèlisation difficile dans des outils génériques :

- Les développeurs OCaml aiment à garder la possibilité de compiler leurs projets aussi bien en bytecode (pour compiler rapidement et obtenir un retour rapide sur leurs modifications) qu'en code natif (pour obtenir une meilleure performance à l'exécution).
- La règle de compilation des implémentations peut générer plusieurs fichiers quand l'interface texte d'une implémentation n'est pas disponible.

La combinaison de ces règles peut poser des problèmes dans un contexte concurrent : ainsi, supposons que le projet est composé de trois fichiers `a.ml`, `b.ml` et `c.ml`, chaque module dépendant du précédent. Ainsi, l'ordonnancement des règles suivantes est incorrect, et produira une erreur de temps en temps :

1. Compilation de `a.ml` en bytecode et en code natif ;
2. Compilation de `b.ml` en bytecode (le compilation en natif de `a.ml` prend plus de temps qu'en bytecode) ;
3. Compilation de `c.ml` en bytecode et de `b.ml` en code natif ;
4. Link en bytecode et compilation de `c.ml` en code natif ;
5. Link en natif.

L'erreur provient du fait que la compilation de `b.ml` en code natif regénère le fichier `b.cmi`, qui avait déjà été généré par la compilation en bytecode. Or, la compilation en bytecode de `c.ml` nécessite la lecture du fichier `b.cmi`, celui-ci pourra être incomplet si la compilation native a commencé à le générer mais n'a pas fini.

3. Description de projet

`ocp-build` est à la fois une bibliothèque permettant d'exploiter des fichiers de description de *paquets*, et un outil pour compiler un ensemble clos de paquets, décrits par ces fichiers. La bibliothèque permet à d'autres outils d'exploiter les descriptions de paquet pour d'autres tâches que la compilation.

Par exemple, notre environnement de développement `TypeRex` l'utilise dans les outils suivant :

Outils de refactoring Le refactoring de code impose de ne pas modifier la sémantique du programme dans son ensemble quand une modification est apportée dans un de ses composants. Cela nécessite donc d'avoir la description de l'ensemble des fichiers d'origine de tous les paquets pour pouvoir répercuter une modification dans un paquet sur l'ensemble des paquets qui l'ont comme pré-requis.

Outils de navigation Comme pour le refactoring, la navigation nécessite une connaissance globale du projet et de ses paquets. Ainsi, si l'utilisateur qui veut visualiser l'interface d'un module, l'outil de navigation doit retrouver le module dans le paquet, ou dans la fermeture transitive de ses pré-requis, puis identifier le fichier d'origine (interface ou implémentation).

Outils de génération de code Certains de nos outils permettent d'utiliser les fichiers objets pour générer du code. Ainsi, `ocp-mli` peut génèrer aussi bien un fichier texte d'interface qu'un fichier texte d'implémentation à partir d'un fichier binaire d'interface. Comme les fichiers binaires ne sont pas toujours générés dans le répertoire d'origine, l'outil doit utiliser la description de projet pour comprendre comment le projet est compilé, et par conséquent où trouver les fichiers binaires d'interface correspondant aux fichiers texte d'origine.

Outils de documentation Nous avons développé un plugin d'`ocamldoc` pour générer de de la documentation avec une interface utilisateur améliorée et qui prend en compte l'organisation en paquets d'un projet. L'outil utilise la description de projet pour appeler `ocamldoc` successivement sur chaque fichier d'interface, avec les chemins à inclure, le pré-processeur et les options `pack` à utiliser. Il utilise aussi `ocamldoc` pour générer les pages d'index des projets à partir de fichiers texte décrivant les paquets et le projet.

Un projet est décrit par deux types de fichiers :

- Des fichiers de description (avec une extension `ocp`) fournissent dans chaque répertoire un ou plusieurs paquets, les fichiers d'origine qui les composent, leurs pré-requis, ainsi que les contraintes spécifiques pour générer les fichiers cibles.
- Un fichier `ocp-build.root` à la racine du projet, fournit la liste des fichiers de description du projet, ainsi que certaines options pour compiler le projet.

3.1. Configuration du projet

Le fichier `ocp-build.root` peut être généré ou modifié automatiquement en utilisant la commande :

```
$ ocp-build -init -scan
```

Cette commande permet de générer un fichier `ocp-build.root` dans le répertoire courant, et de parcourir tous les sous-répertoires pour y trouver les fichiers de description de projet. L'argument `-scan` peut être utilisé séparément pour remettre à jour la liste des fichiers de description.

Plusieurs options peuvent être définies dans la configuration du projet. Si définies, elles sont prioritaires sur les réglages de l'utilisateur.

- L'option `ncores` définit le nombre de processus qui peuvent s'exécuter en parallèle ;
- L'option `digest` indique si des checksums doivent être calculées pour décider de la nécessité d'une recompilation ;
- L'option `autoscan` indique si `ocp-build` doit parcourir les répertoires pour retrouver les fichiers de description à chaque exécution ;

3.2. Description des paquets

Pour bien comprendre le format de description des paquets, nous allons décortiquer l'exemple suivant :

```
pp = [ "camlp4op" ]

begin library "foo"
  files = [ "fooX.ml"; "fooY.mll"; "fooZ.mly"; "foo_c.c" ]
  requires = [ "unix" ]
end
```

```
begin program "bar"
  sort = true
  link += [ "-linkall" ]
  files = [
     "barC.ml" (asmcomp += [ "-inline";  "30"])
     "barB.ml"; "barA.ml" ]
  requires = [ "foo" ; "dynlink" ]
end
```

Ce fichier décrit deux paquets, une bibliothèque `foo` et un programme `bar`. La directive `files` liste les fichiers d'origine qui composent le paquet. La directive `requires` liste les pré-requis, i.e. les autres paquets dont dépend un paquet.

Plusieurs options ici sont intéressantes :

- L'option `pp` indique le préprocesseur qui doit être utilisé pour compiler les fichiers `ml` ou `mli`. Comme toutes les commandes ou arguments dans `ocp-build`, il s'agit non pas d'une chaîne de caractères mais d'une liste de chaîne, ce qui permet d'éviter les ambiguités habituelles avec les espaces et caractères d'échappement.
- L'option `sort` informe `ocp-build` s'il doit trier les modules dans l'ordre de leurs dépendances au moment du *link*. Nous avons pris le parti de désactiver cette option par défaut, pour rappeler aux développeurs OCaml que l'ordre des modules peut avoir de l'importance, si l'initialisation de ces modules comporte des effets de bord.
- L'option `link` permet d'ajouter un argument lors du `link` du programme. `ocp-build` fournit plusieurs options permettant de spécifier des arguments supplémentaires lors des appels aux commandes OCaml :
  - `dep`, `comp` et `link` permettent de spécifier des arguments à utiliser respectivement lors du calcul de dépendances (`ocamldep`), lors de la compilation d'une unité, et lors du link final ;
  - les options `comp` et `link` peuvent être préfixées par `byte` ou `asm` pour indiquer que les arguments doivent être ajoutés respectivement pour la compilation en bytecode ou en code natif ;
  - les option `o`, `asm` et `byte` permettent de spécifier des arguments qui doivent être utilisé aux trois phases de la compilation ;
- Le fichier `barC.ml` est suivi d'options entre parenthèses qui ne s'appliquent qu'à ce fichier ;

On peut finalement remarquer que, pour que la compilation de `bar` réussisse, l'option `-custom` doit être rajoutée et la bibliothèque `unix` doit être fournie au moment du link. `ocp-build` infère automatiquement ces actions à partir des fichiers d'origine (fichier C `foo_c.c`) et de la fermeture transitive des paquets. Par ailleurs, `ocp-build` infère aussi automatiquement les arguments de chemin `-I` à fournir aux compilateur.

Finalement, `ocp-build` fournit une syntaxe élégante pour gérer l'option `-pack` d'OCaml :

```
begin library "foobar"
  files = [
    pack FooBar [
      pack Foo [ "fooA.ml" "fooB.ml" ]
      pack Bar (sort = true) [ "barB.ml" "barA.ml" ]
    ]
  ]
end
```

Ce fichier est moralement équivalent à compiler un fichier `fooBar.ml` contenant le code suivant :

```
module Foo = struct
  module FooA = struct (* fooA.ml *) end
  module FooB = struct (* fooB.ml *) end
end
module Bar = struct (* sort = true ! *)
  module BarA = struct (* barA.ml *) end
  module BarB = struct (* barB.ml *) end
end
```

4. Compilation

Cette section décrit le fonctionnement d'`ocp-build` quand il est exécuté en ligne de commande pour compiler un projet.

4.1. Fonctionnement

1. Lorsqu'il est appelé, `ocp-build` commence par remonter l'arborescence des fichiers à la recherche du fichier `ocp-build.root`. Lorsque celui-ci est trouvé, `ocp-build` se positionne dans son répertoire, d'où toutes les commandes seront évaluées.
2. `ocp-build` charge l'ensemble des fichiers de description de paquets indiqués dans le fichier `ocp-build.root`. Il est possible de lui demander de parcourir systématiquement l'ensemble des répertoires du projet pour mettre cette liste à jour. Par ailleurs, certains fichiers de description de projet sont aussi utilisés pour indiquer les composants déjà installés sur l'ordinateur.
3. `ocp-build` effectue un tri topologique sur les paquets, pour les parcourir dans un ordre compatible avec les pré-requis spécifiés pour chaque paquet ; les paquets dont les pré-requis ne sont pas disponibles, ou qui sont désactivés, sont retirés de cette liste.
4. Pour chaque paquet restant, dans l'ordre topologique, `ocp-build` génère des règles de compilation pour chaque fichier cible ou temporaire ; ces règles ne sont pas activées par défaut.
5. A partir de tous les fichiers cibles, ou de ceux qui ont été spécifiés sur la ligne de commande, `ocp-build` active toutes les règles nécessaires pour obtenir ces fichiers cibles.
6. Pour chaque règle activée, `ocp-build` calcule le nombre de fichiers sources manquant.
7. Le moteur de compilation est ensuite lancé : celui-ci gère un certain nombre de commandes en cours d'exécution en parallèle. Tant que le nombre maximum de commandes parallèles n'est pas atteint, ou quand une commande se termine, il essaie de trouver une nouvelle commande à exécuter, en suivant l'algorithme suivant :
   - si tous les fichiers sources d'une règle activée sont présents, la règle est proposée à l'exécution ;
   - pour chaque règle proposée à l'exécution, `ocp-build` calcule une chaîne de caractère représentant le calcul à effectuer, puis son *hash*. Cette chaîne de caractères tient compte de l'*état des fichiers sources*, de l'*état des fichiers cibles*, ainsi que de la commande à exécuter pour générer les fichiers cibles ;
   - le hash de la règle est comparé a.vec le hash correspondant à l'exécution précédente de la règle, conservé dans un *cache* persistent. Si le hash est identique, les arguments et la commande sont identiques à la dernière application de la règle, et l'exécution de la règle est jugée inutile, les fichiers cibles passent dans l'état présent, et les règles qui en dépendent peuvent être examinées ;

- si le hash est différent, la règle peut être exécutée. Celui-ci crée un répertoire temporaire pour l'exécution de la règle, où les fichiers cibles seront générés, puis lance la commande en tâche de fond ;
- lorsque la commande a fini de s'exécuter, les fichiers générés sont déplacés vers leurs répertoires de destination, un nouvel hash est calculé et placé dans le cache persistent, et les fichiers cibles sont marqués comme présents ;

Dans l'algorithme que nous venons de décrire, le nombre de fichiers sources manquant joue un rôle très important, puisque c'est ce nombre qui à chaque instant décide de la possibilité d'exécuter une règle. Néanmoins, il peut sembler que dans la description précédente, ce nombre ne peut que décroître depuis la création de la règle. Ce n'est pourtant pas le cas : avant de pouvoir compiler un fichier d'interface ou d'implémentation, deux règles sont générées pour calculer ses dépendances. La première règle exécute `ocamldep` en mode abbrégé si le fichier a été modifié, pour mettre à jour la liste des dépendances. La seconde règle charge cette liste de dépendances, et rajoute à la règle de départ de nouveaux fichiers sources, correspondant à ces dépendances, augmentant ainsi dynamiquement le nombre de sources manquant d'une règle.

Un autre aspect important est qu'`ocp-build` ne calcule pas les dépendances en lisant le système de fichiers, comme le fait `ocamldep`. En effet, `ocp-build` calcule à la place une image virtuelle du système de fichiers, qu'il peuple en y rajoutant les fichiers cibles et les fichiers temporaires. Ainsi, le calcul des dépendances peut s'effectuer sans que les fichiers temporaires aient besoin d'avoir été générés.

4.2. Emplacement des fichiers générés

Traditionnellement, `make` génère les fichiers binaires dans le même répertoire que les fichiers textes. `ocamlbuild` effectue lui une copie de l'arborescence des fichiers textes dans un répertoire `_build`, puis génère les fichiers binaires dans cette seconde arborescence.

`ocp-build` utilise une disposition encore différente des fichiers générés. Les fichiers cibles et les fichiers temporaires d'un paquet `package` sont placés dans un répertoire propre à ce paquet `_obuild/package/`. Ce répertoire peut lui-même être divisé en sous-répertoires si le paquet possède plusieurs variantes (cas de paquets dont les versions bytecode et native sont distinctes, ou encore la version *multi-thread*).

Cette organisation présente plusieurs avantages :

- Un même fichier d'origine peut être compilé de plusieurs façons dans des paquets différents, sans que cela ne complexifie la compilation du projet ;
- Dans un même paquet, un fichier peut être compilé de plusieurs façons, tant que les fichiers générés sont placés dans des sous-répertoires différents. Dans la version suivante d'`ocp-build`, nous envisageons de générer plusieurs variantes *classiques* d'un même paquet :
  - Une version de développement, dans laquelle tous les fichiers sont compilés avec les informations de *debug* (argument `-g`), et les optimisations inter-modules entre paquets sont désactivées ;
  - Une version *finale*, dans laquelle les optimisations sont poussées au maximum, avec en particulier les optimisations inter-paquets ;
  - Une version de *profiling*, dans laquelle les informations de profiling sont ajoutées (argument `-p`)

  En particulier, le fait de disposer les fichiers binaires natifs dans des sous-répertoires différents permet d'activer ou de désactiver les optimisations inter-modules entre des paquets différents, ce qui est très difficile à faire par une autre méthode. Notons aussi que le fait de désactiver les optimisations inter-paquets permet de rendre les paquets plus indépendants, et en particulier de supporter la mise à jour binaire des paquets en pré-requis tant que leurs interfaces ne changent pas ;

– Un dernier avantage est un contrôle plus fin de la portée des modules, dans les cas d'utilisation de l'option -pack. Ainsi, dans l'exemple final de la section 3, ocp-build génère l'arborescence suivante après une compilation native :

```
_obuild/foobar/fooBar.{cmi,cmo,cmx,o,cmxa,a}

_obuild/foobar/FooBar/foo.{cmi,cmo,cmx,o}
_obuild/foobar/FooBar/bar.{cmi,cmo,cmx,o}

_obuild/foobar/FooBar/Foo/fooA.{mlmods,cmi,cmo,cmx,o}
_obuild/foobar/FooBar/Foo/fooB.{mlmods,cmi,cmo,cmx,o}

_obuild/foobar/FooBar/Bar/barA.{mlmods,cmi,cmo,cmx,o}
_obuild/foobar/FooBar/Bar/barB.{mlmods,cmi,cmo,cmx,o}
```

On voit que dans cette arborescence, les modules internes sont compilés dans des sous-répertoires portant le nom des modules dans lesquels ils sont inclus. Si on examine la commande utilisée par ocp-build pour compiler le fichier barB.ml :

```
ocamlopt.opt -c -o ${temp}/barB.cmx -for-pack FooBar.Bar
  -I _obuild/foobar/FooBar/Bar
  -I _obuild/foobar/FooBar
  -I _obuild/foobar
  -I .
  barB.ml
```

on peut remarquer que seuls les modules réellement accessibles à l'exécution depuis BarB sont effectivement disponibles à la compilation.
Pour s'en convaincre, on peut ajouter dans le module BarB une référence directe au module FooA. Une telle référence génèrera immédiatement une erreur avec ocp-build, puisque FooA aurait dû être accédé à travers le module Foo (Foo.FooA). Au contraire, avec d'autres gestionnaires de compilation, l'erreur passera inaperçue, et ne sera détectée que quand la bibliothèque sera utilisée, potentiellement après sa diffusion binaire :

```
ocamlopt -for-pack FooBar.Foo -c fooA.ml
ocamlopt -for-pack FooBar.Foo -c fooB.ml
ocamlopt -for-pack FooBar -pack -o foo.cmx fooA.cmx fooB.cmx
ocamlopt -for-pack FooBar.Bar -c barA.ml
ocamlopt -for-pack FooBar.Bar -c barB.ml
ocamlopt -for-pack FooBar -pack -o bar.cmx barA.cmx barB.cmx
ocamlopt -pack -o fooBar.cmx foo.cmx bar.cmx
ocamlopt -o foobar.cmxa fooBar.cmx
ocamlopt -o main foobar.cmxa main.ml
File "main.ml", line 1, characters 0-1:
Error: No implementations provided for the following modules:
         FooB referenced from fooBar.cmx
make: *** [asm] Error 2
```

5. Réalisation

Dans cette section, nous présentons comment `ocp-build` a été implanté, son architecture, les problèmes rencontrés lors de sa conception, et quelques résultats expérimentaux.

5.1. Architecture

L'architecture d'`ocp-build` se décompose en trois parties :
- Une bibliothèque `ocpbuild-config`, qui permet de lire les fichiers `ocp`, composant un projet dans sa globalité, et de générer une représentation abstraite manipulable facilement par le programme.
- Une bibliothèque `ocpbuild-engine`, qui contient le moteur de compilation, qui prend l'ensemble des règles de compilation, un ensemble de cibles à générer, et exécute toutes les commandes dans l'ordre des dépendances.
- Le programme principal, qui lit la description de projet et génère les règles de compilation correspondante, avant de laisser la main au moteur de compilation pour obtenir le résultat.

5.2. Le moteur de compilation

Le moteur de compilation doit exécuter les règles nécessaires à la création d'un certain nombre de cibles. Actuellement, l'ensemble des règles nécessaires doivent être présentes, même si les dépendances entre elles vont être modifiées dynamiquement.

Pour effectuer ce travail, l'ensemble des règles de compilation forme un graphe orienté :
- Les nœuds du graphe sont les fichiers et les règles de compilation ;
- Les arcs du graphe sont les relations de dépendances, d'une règle vers ses sources, et les relations de générations, d'une cible vers la règle qui la génère.

Le moteur de compilation commence par parcourir ce graphe, à partir des cibles qu'il doit générer. Toutes les règles parcourues sont *activées*. Ensuite, le moteur divise les règles activées en deux ensembles : les règles *prêtes*, dont toutes les dépendances sont à jour, et les règles *en attente*, dont certaines dépendances ne sont pas encore à jour. Le moteur maintient pour chaque règle un compteur des dépendances qui ne sont pas encore à jour.

Quand une règle est exécutée, le compteur des autres règles peut aussi bien croître que décroître :
- Pour tous les fichiers cibles d'une règle qui vient d'être exécutée, on peut décroître le compteur de toutes les règles dont ils apparaissent dans les fichiers sources ;
- Une règle de lecture des dépendances d'un fichier source va rajouter des sources aux règles qui utilisent ce fichier source, ce qui augmente la valeur de leurs compteurs ;

Lorsqu'une règle est prête, le moteur doit décider s'il exécute la règle, ou si l'exécution est inutile. L'exécution est jugée inutile si la commande ne va pas fournir un résultat différent de l'exécution précédente de la règle. Traditionnellement, des outils comme `make` se contentent de comparer la date de modification des fichiers cibles avec la date de modification la plus récente des fichiers sources. Si tous les fichiers cibles sont plus récents, `make` suppose que la commande a été exécutée après la dernière modification, et qu'il n'est donc pas utile de la ré-exécuter. `ocp-build` utilise un algorithme très différent :
- Pour chaque fichier source, le moteur extrait son état au moment de l'exécution de la règle. Cet état peut prendre deux formes différentes :
  - Par défaut, `ocp-build` extrait la date de dernière modification, le numéro de partition du fichier, et le numéro d'inode du fichier.
  - Avec l'option `digest` activée, `ocp-build` calcule une signature du contenu du fichier (via un hash `sha1`), et utilise cette signature comme état.

  La première solution présente l'avantage que l'état est obtenu très rapidement (par un `lstat` du fichier), mais toute modification du fichier provoque un changement d'état, même si son contenu

Architectures cibles	Commandes exécutées	Fichiers générés	Degré de parallèlisme 1	2	3	4	5	6	Gain obtenu
Bytecode uniquement	2147	2375	56.7s	31.1s	24.9s	23.1s	23.1s	23.1s	-59%
Natif uniquement	2120	2909	76.9s	44.1s	36.1s	32.5s	33.1s	33.2s	-57%
Bytecode et natif	2743	3676	111.1s	65.2s	52.8s	49.3s	49.3s	49.4s	-55%

TABLE 1 – Temps de compilation de `TypeRex` en fonction du degré de parallèlisme, sur une machine quatre cœurs, et de l'architecture demandée (bytecode et/ou code natif). Le gain croît jusqu'à -59% pour un degré de 5, et varie peu suivant l'architecture demandée.

n'a pas changé ; au contraire, la seconde solution demande plus de calculs de signatures, mais l'état ne change que si le contenu du fichier change.

- Ensuite, pour chaque règle, le moteur calcule une représentation textuelle de la règle. Cette représentation contient :
  - Les noms des fichiers cibles qui vont être générés ;
  - Les noms des fichiers sources, ainsi que leur état au moment de l'exécution de la règle ;
  - Les différentes commandes qui vont être exécutées.

  Une fois cette représentation calculée, une nouvelle signature est générée ;
- Finalement, le moteur compare la signature de la représentation de la règle à la signature de la dernière exécution de la règle. Si la signature est différente, la règle doit être exécutée à nouveau. Pour permettre la comparaison avec les signatures des anciennes règles, `ocp-build` maintient un cache persistent de ces signatures. Bien-sûr, à sa première exécution, la cache n'existant pas, `ocp-build` infère qu'il doit ré-exécuter toutes les règles.

Cet algorithme présente plusieurs particularités intéressantes par rapport à celui de `make` :

- Si la commande pour générer un fichier change (par exemple, parce que les arguments sont différents, ou l'ordre des répertoires a changé), sa signature change aussi et la règle est ré-exécutée ;
- Si la liste des sources change sans que les fichiers eux-mêmes ne changent (rajout d'un module dans un répertoire cachant le même module dans un répertoire suivant), la commande est encore ré-exécutée ;
- En mode `digest`, la regénération d'un fichier n'implique pas la recompilation des fichiers qui en dépendent si le contenu du fichier n'a pas changé par rapport à la dernière exécution.

5.3. Expérimentation

Pour tester l'efficacité de la compilation en parallèle avec `ocp-build`, nous avons compilé l'ensemble des fichiers de `TypeRex` en changeant le degré de parallelisme, sur une machine à quatre cœurs. La Table 1 montre les résultats obtenus. Le compilation est à peu près deux fois plus rapide dans tous les cas (bytecode, code natif, ou les deux). Ces résultats peuvent surprendre dans la mesure où, intuitivement, les contraintes plus faibles lors de la compilation en bytecode auraient pu faire espérer un gain bien supérieur pour le code bytecode que pour le code natif. Le fait que le facteur soit décorélé du nombre de cœurs n'est par contre pas surprenant : en particulier, il existe dans `TypeRex` quelques bibliothèques qui sont utilisées par de très nombreux composants, et de nombreux modules ne possèdent pas d'interface. Aussi, ces composants limitent le degré de parallèlisation réelle obtenue.

6. Conclusion

Dans cet article, nous avons présenté `ocp-build`, notre outil de gestion de projet. `ocp-build` permet de décrire un projet en composants séparés dans un langage de description intuitif pour

OCaml. Il permet ensuite de les compiler en profitant au maximum d'une compilation fiable, à la fois en mode incrémentale et en mode parallèle.

Par ailleurs, les fichiers de descriptions de projet d'`ocp-build` présentent de nombreux avantages : d'une part, ils améliorent la lisibilité des projets, en décrivant de façon concise l'architecture des composants du projet ; d'autre part, ils décrivent à la fois la façon de compiler les composants, et la façon d'utiliser ensuite ces composants pour construire de nouveaux composants. Parce qu'ils décrivent les modules d'un composant, il peuvent aussi permettre d'inférer automatiquement les dépendances entre composants. Aussi, l'ensemble de ces propriétés pourraient en faire de bon remplaçant pour les fichiers `META` d'`ocamlfind`.

Références

[1] Coutts, D., Potoczny-Jones, I., and Stewart, D. Haskell : batteries included. In *Proceedings of the first ACM SIGPLAN symposium on Haskell* (New York, NY, USA, 2008), Haskell '08, ACM, pp. 125–126.

[2] Hickey, J., and Nogin, A. Omake : Designing a scalable build process. In *Fundamental Approaches to Software Engineering, 9th International Conference, FASE 2006* (2006), L. Baresi and R. Heckel, Eds., vol. 3922, Springer, pp. 63–78.

[3] Leroy, X. *The Objective Caml System : Documentation and User's Manual*, 1996. With Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. Availiable from `http://www.ocaml.org/`.