OGIT

Programmation fonctionnelle - 2022-2023

Le but de ce projet est d'implémenter le noyau minimal d'un logiciel ressemblant à git, c'est-à-dire permettant de gérer les versions d'un ensemble de fichiers. Ce git ne fonctionnera qu'en local pour un seul utilisateur.

Concepts git

Si vous n'êtes pas familier avec git vous pouvez vous familiariser avec les concepts de dépôt, de commit, de checkout, de branche, et de merge sur ce site: https://learngitbranching.js.org/ (seuls les 3 premiers niveaux de la séquence d'introduction vous seront utiles ici). Vous pouvez aussi simplement utiliser la commande git en ligne de commande pour comprendre comment elle marche (si elle n'est pas installée sur votre ordinateur, installez-là!)

Nota:

- *Nous ne définirons pas de commande "ogit branch", pour nous les branches sont uniquement identifiées par leur hash.*
- *Nous utiliserons un hash MD5 (module Digest)*
- *Tous les fichiers présents dans le dépôt dont le nom ne commence pas par un `.` seront automatiquement "suivis" par ogit (pas de commande ogit add, ou de fichier .ogitignore, par exemple).*

Quelques points importants avant de commencer

Travailler en équipe

Vous pouvez faire ce projet seul, en binôme, ou en trinôme. L'utilisation de `git` ou d'un autre gestionnaire de versions est recommandée! Vous apprendrez à la fois à quoi cela sert, et comment cela fonctionne. Rappelons la règle suivante: vous ne devez pas échanger de fichiers avec d'autres personnes que votre équipe. Les discussions sur moodle ou discord sont encouragées, c'est très bien si vous posez des questions, et c'est très bien si vous aidez quelqu'un qui a posé une question à la place de l'enseignant (si ce que vous dites n'est pas correct, on corrigera). Mais restez au niveau de la langue française (ou d'un dessin), ne montrez pas de code (occasionnellement, 1 ou 2 lignes de codes peuvent être divulguées si votre question ou votre réponse n'a pas été comprise, mais évitez autant que possible). Enfin, si

vous utilisez un dépôt dans le cloud (bitbucket, github, etc) paramétrez votre dépôt pour qu'il ne soit pas public et que seul votre équipe y ait accès. Nous utilisons un logiciel de détection de plagiat et des points seront retirés aux plagieurs mais aussi aux plagiés (sans chercher à faire de distinction) en cas de plagiat détecté, même partiel.

Découpage du projet, prise en main de `dune`

Pour démarrer le projet il vous faut installer git sur votre machine (apt-get install git, brew install git, etc), installer dune (opam install dune), et cloner le dépôt git contenant tous les fichiers pour démarrer le projet ici: https://github.com/etiloz/projet-pf-2022-2023-ogit . Comme vous pouvez vous en rendre compte, le projet contient de nombreux fichiers dans divers répertoires. Il y aura, en plus de l'exécutable final, de nombreux tests qui permettent d'évaluer ce que fait le code, ainsi qu'un dépôt "jouet" pour tester ogit.

Le projet est prévu pour être compilé et testé avec le gestionnaire de projet OCaml `dune`. Pour permettre d'évaluer vos projets, veuillez ne pas changer l'arborescence des répertoires ni les noms de fichiers.

Voici une description rapide de l'arborescence

- `ogit` : contient l'ensemble du projet géré par dune (Consultez la documentation de dune pour plus de précisions)
- `ogit/lib` : contient le code que vous devez écrire, ainsi qu'un peu de code fourni
- `ogit/test`: contient tous les tests fournis
- `repo`: contient un dépôt "jouet" utilisé pour les tests. Les fichiers "versionnés" (qui constituent le dépôt) sont ceux dont le nom ne commence pas par un `.`, présents à la racine de `repo/` ou dans des sous-répertoires
- `repo/.ogit` : le répertoire dans lequel ogit va stocker les objets et les commits

Pour une première prise en main, exécutez depuis le répertoire `ogit` la commande `dune runtest test/hello`. Vous allez voir que le test échoue. Pour que le test passe, il faut que l'affichage généré par `test/hello/test_hello.ml` corresponde à ce qui est spécifié dans le fichier `test/hello/test_hello.expected`. Ne touchez pas le fichier `test/hello/test_hello.ml` ni le fichier `test/hello/test_hello.expected` (de manière générale, ne modifiez pas les tests qui vous sont fournis), mais corrigez plutôt le fichier `ogit/lib/hello/hello.ml`. Ré-exécutez la commande `dune runtest test/hello` et vérifiez que cette fois-ci le test passe.

Travail à rendre

Vous aurez à rendre tous les fichiers qui constituent votre projet:

- les fichiers fournis que vous aurez complétés dans le répertoire `ogit/lib/`
- les fichiers `AUTHORS` et `README` que vous aurez complétés. Le fichier `README` explique ce que vous avez fait, ce qui marche, les bugs connus, et ce que vous n'avez pas eu le temps de faire. Il n'y a pas forcément besoin de raconter sa vie dans ce fichier, c'est surtout pour pouvoir confronter ce que vous pensez de votre projet à ce qu'en disent les tests.
- il n'est pas nécessaire de rendre le répertoire `test/` fourni. Si vous le rendez, le code qu'il contient ne sera pas évalué.

Créez une archive avec tous ces fichiers et mettez-la dans la boite de dépôt Moodle.

Date limite de dépôt des projets: 14 décembre minuit. 2 points de pénalité seront appliqués par jour de retard.

Repertoire .ogit

Le répertoire `.ogit` est un répertoire qui sera maintenu par votre programme ogit à la racine du dépôt (`repo/.ogit`). Ce répertoire contient toutes les données gérées par votre logiciel. Il contient :

- Un fichier “HEAD” qui contient le hash du dernier commit (ou des derniers commits en cas de merge)
- Un répertoire “logs” qui contient l’ensemble des commits. Chaque commit est stocké sous la forme d’un fichier qui contient les informations relatives à un commit et dont le nom est le hash (MD5) de son contenu.
- Un répertoire “objects” qui contient l’ensemble des objets versionnés. Chaque objet versionné est stocké sous la forme d’un fichier qui contient cet objet et dont le nom est le hash (MD5) de son contenu.

Les objets versionnés

OGit considère deux types d’objets versionnés représentés par des fichiers du répertoire `repo/.ogit/objects/:

- Les fichiers texte : qui contiennent un texte
- Les répertoires, qui contiennent un ensemble d’objets versionnés.

Un répertoire est une liste de noms associés à des hashs.

Par exemple, un répertoire “repo” contenant les trois fichiers suivants :

Name	**Type**	**Hash**

main.ml	text	94daecdffe4003a70f02ee8989295b32
foo	directory	a94a8fe5ccb19ba61c4c0873d391e987
config	text	dfba7aade0868074c2861c98e2a9a92f

donnera lieu à un objet versionné stocké sous la forme d'un fichier texte dont le contenu est

```
main.ml;t;94daecdffe4003a70f02ee8989295b32
foo;d;a94a8fe5ccb19ba61c4c0873d391e987
config;t;dfba7aade0868074c2861c98e2a9a92f
```

Sans '\n' final.

et dont le nom est ".ogit/objects/3b605b2ff337efc5bf4213e76dcd9169" (le MD5 du contenu). Il y a donc 4 fichiers dans le répertoire "objects" correspondant à 4 hash. Si on modifie le fichier "main.ml" et que l'on commit nos modifications, il y aura 2 nouveaux fichiers dans "objects" :

- La nouvelle version de "main.ml" (sous son nouveau hash)
- La nouvelle version du répertoire "project" puisque son contenu, et donc son hash, sont modifiés

Les 4 fichiers originaux (correspondants à "foo" et "config" ainsi que les anciennes versions de "main.ml" et de "repo") seront toujours présents et inchangés.

Git ne prend en compte que le contenu des fichiers, représenté par un hash (MD5). Deux fichiers ayant le même contenu auront le même hash et ce contenu ne sera donc stocké qu'une seule fois.

Le module Objects

Votre première mission va être d'écrire le module Objects permettant de lire et écrire les fichiers dans le répertoire `repo/.ogit/objects` et d'avoir une structure de données qui permette de représenter les objets versionnés.

Compte tenu de la nature des objets versionnés, on adopte la définition suivante pour le type Objects.t
(cf fichier `ogit/lib/objects.mli`)

```
type t =
| Text of string
| Directory of (string * bool * Digest.t * t) list
```

Un objet qui est un fichier texte est représenté dans OCaml par une chaîne de caractères. Un objet qui est un répertoire est représenté par la liste des lignes de son fichier, mais aussi pour chaque ligne on ajoute récursivement l'objet vers lequel pointe le hash. Cela devrait vous rappeler la structure d'arbre d'arité variable vue en cours. Le booléen est à vrai si le drapeau est sur `d` (i.e. le fichier est un répertoire).

Complétez le fichier `objects.ml` de sorte à satisfaire la signature `objects.mli` et à passer les tests exécutables avec la commande `dune runtest test/objects`. Lisez bien les descriptions des fonctions dans le fichier `objects.mli`. En particulier, notez bien que les fonctions seront amenées à être appelées par un programme qui s'exécute à la racine du dépôt (répertoire repo/).

Quelques modules utiles (lire la doc pour voir les fonctions qui peuvent vous servir):

- Digest : fonction de hachage MD5
- Sys : fonctions basées sur le système d'exploitation
- Unix : fonctions pour notamment récupérer l'heure courante et la convertir en h/m/s-j/m/a
- String : fonctions de manipulation de chaînes de caractères
- In_channel et Out_channel : fonctions d'entrée-sorties avancées (gestion facilitée des exceptions)

Les commits et le répertoire logs

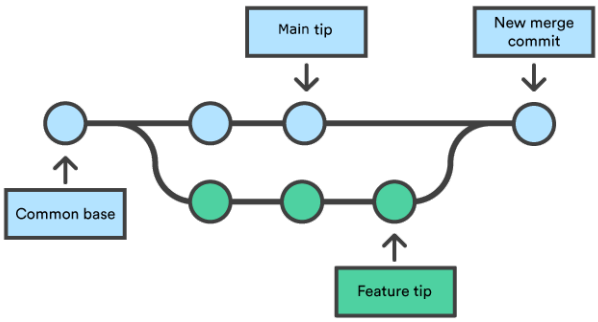
On s'intéresse maintenant au répertoire `repo/.ogit/logs` et aux commits qu'il contient. Chaque fichier de ce répertoire correspond à un commit. Un commit contient les informations suivantes:

- une description
- un parent - ou plusieurs si il s'agit d'un merge
- le hash de l'état du répertoire racine.

Vous devez pouvoir effectuer les actions suivantes :

- Obtenir les informations relatives à un commit à partir de son hash (i.e. lire le fichier ".ogit/logs/<hash>" correspondant)
- Ajouter un commit (i.e. créer le fichier ".ogit/logs/<hash>" correspondant. Dans le cas d'un merge cela doit provoquer une erreur si l'un des parents est l'ancêtre d'un autre)

Par exemple, le schéma ci-dessous correspond à 7 commits.



(c) Atlassian

Tous les commits ont pour ancêtre le commit “Common base”. Les deux commits “Feature tip” et “Main tip” sont dans des branches séparés et ne sont donc pas ancêtres l’un de l’autre. Le commit “New merge commit” a pour parent (et donc ancêtre) ces deux commits.

Le format du contenu d’un fichier représentant un commit est le suivant :

```
94daecdffe4003a70f02ee8989295b32;a94a8fe5ccb19ba61c4c0873d391e987
16:18:40-28/11/2022
Merge bug fix #342
3b605b2ff337efc5bf4213e76dcd9169
```

Sans ‘\n’ final.

C'est-à-dire la liste de hash du (ou des) commit(s) parent(s), la date de ce commit, la description du commit et le hash de la version correspondante du répo.

Complétez le fichier `ogit/lib/logs.ml` et exécutez les tests avec la commande `dune runtest test/logs`.

Commandes ogit

A l’exception de “ogit init”, chacune des actions suivantes peut être appelée depuis la racine d’un dépôt ogit (qui contient donc un répertoire “.ogit”) alors appelé arbre de travail.

- **ogit init** : crée un répertoire “.ogit” dans le répertoire courant avec les sous répertoire “objects” et “logs” ainsi qu’un premier commit vide.
- **ogit commit “<description>”** : parcourt récursivement l’arbre de travail et ajoute tous les nouveaux états rencontrés, puis ajoute le commit correspondant avec pour parent le commit HEAD. *Les fichiers dont le nom commence par un point (e.g. “.mvn”) sont ignorés.*
- **ogit log** : affiche les informations concernant le dernier commit courant (HEAD) et tous ses ancêtres.

- **ogit checkout <hash>** : positionne le répertoire courant dans l'état correspondant au commit <hash>. Met à jour les fichiers dans l'arbre de travail pour correspondre à la version spécifiée. Met aussi à jour HEAD comme étant cette version.
- **ogit merge <hash>** : fusionne les modifications du commits nommés dans l'état actuel. *Si le merge se passe sans conflit (voir ci-dessous), un commit est créé et la HEAD est actualisée. Sinon, un message d'information est affiché et le conflit doit être résolu avant de pouvoir effectuer un nouveau commit.*

Git merge I

Lorsque que l'on merge l'état d'un commit X à l'état actuel (le contenu du répertoire repo au moment où on demande le merge), les modifications concurrentes doivent être fusionnées.

- Si un fichier "fich" est présent dans X mais pas présent dans l'état actuel, on l'ajoute dans l'état actuel
- Si un fichier "fich" est présent dans X et dans l'état actuel, avec le même contenu il ne se passe rien
- Si un fichier "fich" est présent dans X et dans l'état actuel, mais avec des contenus différents, on obtient un **conflit**. Deux versions du fichier sont créés : "fich..cl" (version "locale") et "fich..cr" (version "remote" pour X)

Si il n'y a pas de conflit, un commit avec deux parents est immédiatement produit. Sinon, le fichier HEAD est mis à jour avec deux parents mais tant que le conflit n'est pas résolu (présence d'un fichier "..cl" ou "..cr"), une tentative de "git commit" ou "git merge" doit provoquer une erreur. Une fois le conflit résolu, le prochain commit ayant lieu aura deux parents (ou plus si un autre "git merge" est effectué entre temps).

Better Hash

Dans les commandes git checkout et merge, plutôt que d'utiliser les hash MD5 en entier on devrait pouvoir utiliser un préfixe de ces hash. Par exemple, `371a49` à la place de `371a49c846b6b5345dc6753ca664393b`. Faites en sorte que les commandes git puissent utiliser tout préfixe (de taille minimum 4). Si il y a ambiguïté, i.e. un préfixe correspondant à deux hash, la commande doit afficher une erreur.

Git merge II

Le merge "Git merge I" n'est vraiment pas malin. Pour obtenir un meilleur résultat il faut se référer à l'ancêtre commun ("Common base" dans le schéma ci-avant). Pour chaque fichier présent dans l'état actuel L ou dans l'état que l'on fusionne R, on va agir en fonction des modifications apportées depuis l'ancêtre commun.

Nota : il est facile de voir si un fichier a été modifié grâce à son hash.

Répertoires

R\L	Identique	Modifié (ou ajouté)	Supprimé
Identique	RAS	modifié	supprimé
Modifié (ou ajouté)	modifié	merge récursif	merge récursif
Supprimé	supprimé	merge récursif	RAS

Explications:

- Si le répertoire est resté identique par rapport à l'ancêtre commun dans une des versions (R ou L) et a été modifié (resp. supprimé) dans l'autre version il faut que l'état actuel devienne celui mis à jour (modifié ou supprimé).
- Si le répertoire a été modifié dans une des versions (R ou L) et modifié ou supprimé dans l'autre version, il faut faire un merge récursif, c'est à dire garder le répertoire et faire un merge de chaque fichiers présent dans les deux versions du répertoire

Fichiers texte

R\L	Identique	Modifié	Supprimé
Identique	RAS	modifié	supprimé
Modifié	modifié	three-way-merge	modify/delete conflict
Supprimé	supprimé	modify/delete conflict	RAS

Explications:

- Si le fichier texte a été modifié dans une des versions et supprimé dans l'autre il faut prendre la version modifiée et changer le nom du fichier en "fich..mdconflict".
- Si le fichier a été modifié dans les deux versions vous devez effectuer un three-way-merge https://en.wikipedia.org/wiki/Merge_(version_control)#Three-way_merge. Nous vous conseillons très fortement d'utiliser un outil externe (diff3 ou kdiff3) pour ce faire.

Comme pour la version simplifiée du "git merge", si il n'y a pas de conflit un commit avec deux parents est immédiatement produit, sinon, le fichier HEAD est mis à jour et les conflits doivent être résolu (y compris les conflits textuels).

Log Graph

Affichez le log pour représenter les différentes branches à la manière de "git log --graph".

```
*   4a904d7 : initial commit
|\
* | 81a3e0d : updated packfile code to recognize index v2
```

```
| *   dfeffce : Appropriate time-zone test fix from halorgium
| |\
| * | c615d80 : fixed a log issue
| | * 23f4ecf : Clarify how to get a full count out of Repo#commits
| |/
| *   9d6d250 : merge in bryces changes and fixed some testing issues
|/
*   e1e8989 : merge in master branch
```