Principia Softwarica: The Version Control System (ocaml)git

version 0.1

Yoann Padioleau
yoann.padioleau@gmail.com

with code from
Yoann Padioleau

November 3, 2025

Contents

Chapter 1

Introduction

The goal of this book is to explain with full details the source code of a version control system.

1.1 Motivations

Why a version control system (VCS)? Because I think you are a better programmer if you fully understand how things work under the hood, and a VCS is one of the tools a programmer uses the most. Indeed, programmers use VCSs, which manage efficiently changes to a file or set of files, to manage the vital product of their labour: source code.

VCSs allow to store and retrieve past versions of a file, and to record who made each change, when, where, and why. VCSs are mostly used by programmers but you can use a VCS to manage any text-based documents (e.g., the source of this book), configuration files, and even binaries.

For projects with a single developer, a VCS is useful to keep track of changes: it allows the programmer to easily go back to past versions if he messed things up. For projects with multiple developers, a VCS is almost mandatory: it allows programmers to collaborate with each other and even to work on and modify the same files concurrently.

Here are a few questions I hope this book will answer:

- What are the fundamental concepts of a VCS? What are the core algorithms behind a VCS?
- How does a VCS store efficiently multiple versions of a file?
- How does a VCS represent changes to a file, or changes to a set of files? How does a VCS represent the addition, deletion, or renaming of a file?
- How does a VCS allow multiple users to work on and modify the same files at the same time? How can a VCS help coordinate the actions of multiple users?
- How does a VCS support parallel developments in a project? How can some developers work on the main release, some developers on experimental features, and others on fixing bugs on previously shipped releases all at the same time? What is a branch? How does a VCS reconcile multiple branches?
- What is a merge algorithm? When is a merge safe? What is a merge conflict? How does a VCS resolve conflicts?

1.2 The version control system Git (and `ocamlgit`)

I will explain in this book the code of the VCS `ocamlgit`[1], which contains about 6200 lines of code (LOC). As its name suggests, `ocamlgit` is written in OCaml and is a clone of the popular VCS Git[2], which is written in C.

As opposed to most books in Principia Softwarica, I could not choose a Plan 9 program for this book because there are no Plan 9 VCSs. There are many open source UNIX VCSs with different user interfaces, storage strategies, concurrency models, or features. I will present a few of those VCSs in Section 1.3. For this book, I decided to base the presentation on Git because Git is one of the most popular VCSs in the open source community. Moreover, when coupled with the hosting website GitHub[3], Git makes it really easy for people to collaborate with each other.

However, Git is a rather large project with more than 200 000 LOC. It is impossible to present all this code in a book of a reasonable size. There are a few clones of Git written in higher-level languages than C, for example, Dulwich[4] written in Python with only 16 000 LOC. I could have based this book on Dulwich, but this would introduce another language in the Principia Softwarica book series in addition to C (used in most of the books) and OCaml (used in the editor, web browser, and code generators books). This would also in turn require to present the code of the Python interpreter, which contains more than 170 000 lines of C code. Instead, I decided to port the Python code of Dulwich to OCaml, resulting in `ocamlgit`, and to present the code of `ocamlgit`.

1.3 Other version control systems

Here are a few VCSs that I considered for this book, but which I ultimately discarded:

- The Revision Control System (RCS) [Tic85][5] was the first open source VCS. Like its predecessor, the Source Code Control System (SCCS) [Roc75][6], RCS uses *deltas* to store efficiently the past versions of a file (Section 16.4 will present the Diff algorithm to compute those deltas). However, RCS improves over SCCS by using *reverse-deltas* to enable the user to also retrieve quickly the last version of a file (a recurring operation).

  RCS is still a reasonable choice to manage a small project with a single developer or a small set of developers who can share access to a common directory (for example, by using NFS and symbolic links). However, the use of *locks* in RCS to forbid multiple developers to modify the same file at the same time and its focus on individual files make RCS inadequate for large projects with many independent developers. Moreover, even if RCS is very limited compared to modern VCSs like Git, RCS still contains more than 17 500 LOC.

- The Concurrent Versions System CVS [Gru86][7] was the most popular VCS for over a decade before distributed VCSs (e.g., Git) took over. CVS introduced a few important innovations. First, CVS was designed to operate on a set of files at once[8], with files possibly organized in a tree. With CVS, you can group together related changes to a set of files and you can easily retrieve past versions of a whole project. Secondly, CVS allows multiple users to work on and modify the same files concurrently (hence the 'C' in CVS). CVS relies on the `merge` program instead of locks (Section 16.5 will present the Diff3 algorithm which underlies the `merge` program). Finally, CVS introduced later in 1995 a *client/server* architecture where a *repository* could be stored on a remote machine. When coupled with the free hosting site Sourceforge[9], CVS allowed developers to easily start and collaborate on new projects.

[1] https://github.com/aryx/plan9-ocaml/tree/master/version_control
[2] https://git-scm.com/
[3] https://github.com
[4] https://www.dulwich.io/
[5] https://www.gnu.org/software/rcs/
[6] http://sccs.sourceforge.net/
[7] http://www.nongnu.org/cvs/
[8] RCS can operate on multiple files by using shell wildcards, but it was not designed for it.
[9] https://sourceforge.net

CVS started as a few shell scripts using RCS as a backend. However, today CVS is a very large C program with more than 80 000 LOC. The backend file `src/rcs.s` contains already 9000 LOC.

- Git [CS14][10] was originally created by Linus Torvalds in 2005 to manage the source code of the Linux kernel. It followed a series of *distributed VCSs* (e.g., TeamWare, BitKeeper, Arch, Darcs, Monotone) designed to overcome the main limitations of CVS (a centralized VCS). Because a distributed VCS (DVCS) does not require access to a central repository, it allows developers to work independently offline. A DVCS makes it also easy and cheap to create *branches* representing parallel developments, and to reconcile later those branches (see Section 2.1.5 for more discussions on the advantages of distributed VCSs over centralized VCSs).

  As I explained in Section 1.2, Git is a large program with more than 200 000 LOC spread over more than 400 files (not including the tests, GUIs, and extra contributions). The first version of Git was small and contained only 1000 LOC[11]. However, it contained only a few low-level commands that were not easy to use. This forced programmers to develop and use the extra program `cogito`[12], which provided an extra layer of higher-level commands (called *porcelain*), but added many more lines of code.

- Mercurial [O'S09][13] is another popular DVCS started also in 2005 by another Linux kernel programmer: Matt Mackal. It is mostly written in Python and relies only on a few C files for critical operations. Its code is arguably easier to understand than Git but it still contains more than 100 000 LOC spread over more than 170 files (not including the tests, extensions, and extra contributions).

Figure 1.1 presents a timeline of major VCSs. `ocamlgit` is not as efficient and complete as Git or Mercurial, but it provides almost all of the essential features of those programs with more than an order of magnitude less code (6500 LOC).

1.4 Getting started

To play with `ocamlgit`, you will first need to install it by following the instructions at https://github.com/aryx/plan9-ocaml/wiki/Install[14]. Once installed, you can test `ocamlgit` under Plan 9 or UNIX with the following commands:

```
1 $ cd /tests/
2 $ mkdir hello
3 $ cd hello/
4 $ ocamlgit init
5 Initialized empty Git repository in /tests/hello/.git/
6 $ echo "Hello Git" > hello.txt
7 $ ocamlgit add hello.txt
8 $ ocamlgit commit -m "first commit" --author "pad <todo@todo>"
...
```

The command in Line 4 initializes a new *repository* by creating the appropriate metadata in the `/tests/hello/.gi` subdirectory. Line 7 then adds the new file `hello.txt` to the *staging area* of Git (I will explain later in Sec-

[10] https://git-scm.com
[11] https://github.com/git/git/commit/e83c5163316f89bfbde7d9ab23ca2e25604af290
[12] http://git.or.cz/cogito/
[13] https://www.mercurial-scm.org/
[14] As you follow the document in the Wiki, you will see that you will need a VCS program (`git`) to get the source of another VCS program (`ocamlgit`). This is similar to bootstrapping issues in compilers. However, for VCSs an easy way to avoid those issues is to provide instead an archive file of the source (e.g., `ocamlgit-0.1.tar.gz`).

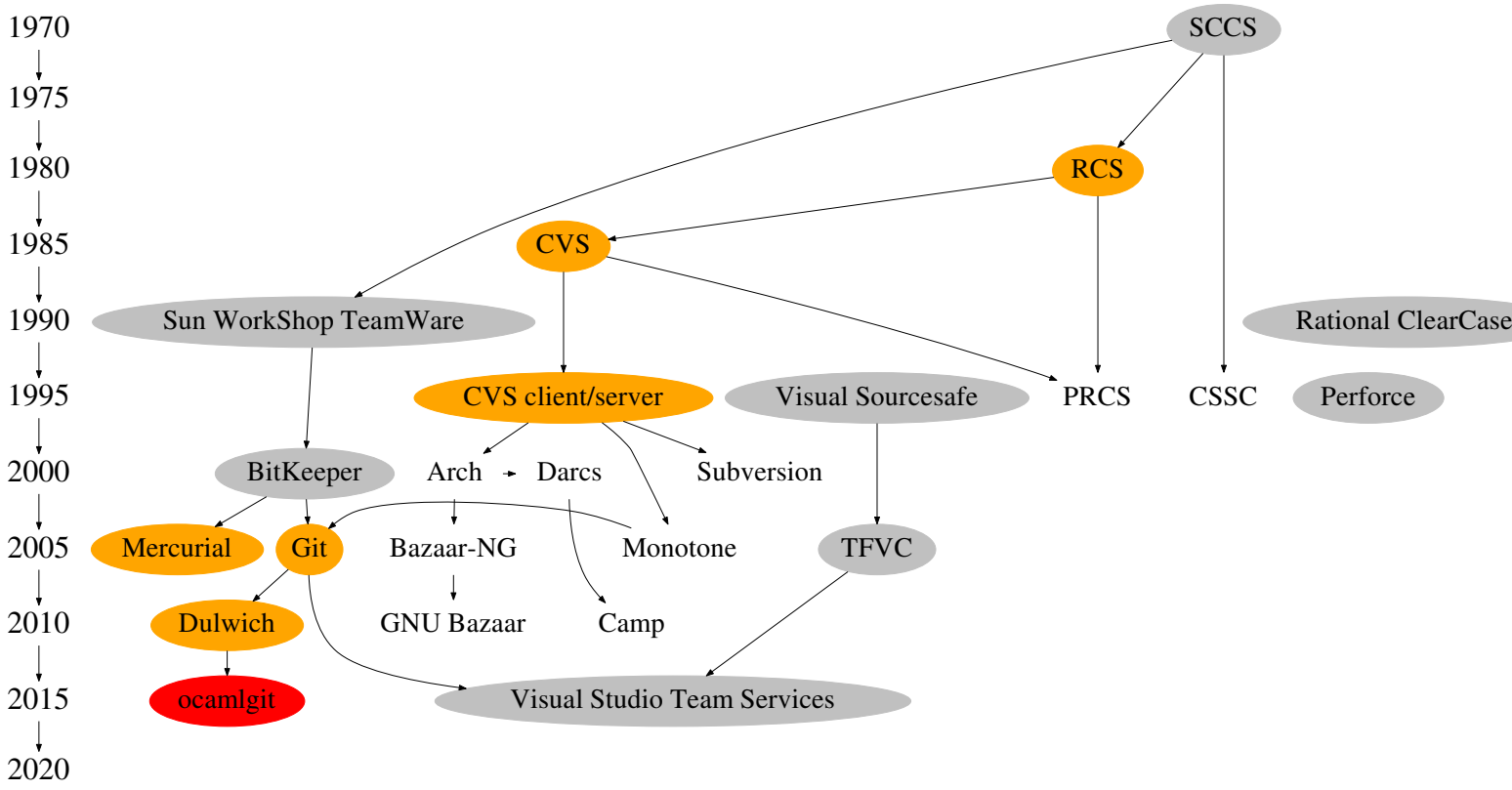

Figure 1.1: Version control systems timeline

tion 2.3.3 what is a staging area). Finally, Line 8 *commits* what was staged to the repository and records this commit with the message "first commit" and the author "pad"[15].

In the rest of this document, I will often abbreviate the command `'ocamlgit'` by using instead the command `'git'`. You can even define `'git'` as an alias for `'ocamlgit'` in your shell. Indeed, `ocamlgit` uses the same command-line interface than the program `git`. Almost all `ocamlgit` commands are valid `git` commands[16]. When the interfaces differ, I will explicitely use `'ocamlgit'`.

1.5 Requirements

Because most of this book is made of OCaml source code, you will need to know the OCaml programming language [LDF+16] to understand it. The code of `ocamlgit` uses only the core language of OCaml, and none of its advanced features (e.g., functors, objects, labels, polymorphic variants, GADTs), thus a knowledge of Caml [WL99], Standard ML [Pau96], or any of the dialect of ML (e.g., F# [Har08]) should be enough to understand this book.

You do not need to know Git, or more generally any VCSs, to understand this book. However, if while reading this book you have specific questions on the interface of Git, I suggest you to consult the man pages of Git[17], or any of the books on Git (e.g., [CS14]).

[15] This is my nickname

[16] The reverse is not true. `ocamlgit` implements only a subset of the commands supported by `git`.

[17] https://git-scm.com/docs

1.6 About this document

This document is a *literate program* [Knu92]. It derives from a set of files processed by a tool, `syncweb` [Pad09], generating either this book or the actual source code of the program. The code and its documentation are thus strongly connected.

1.7 Copyright

I wrote most of the code in this document.

⟨*copyright ocamlgit* 15a⟩≡ (180 177c 176 168 167b 166b 161a 160b 157 156 155 154 153 152 151 150b 149 148b)

```
(* Copyright 2017 Yoann Padioleau, see copyright.txt *)
```

The code is licensed under the GNU Lesser General Public License version 2.1 as published by the Free Software Foundation.

As I said in Section 1.2, most of the code of `ocamlgit` is a port in OCaml of Python code from Dulwich, which is governed by the following copyright:

⟨*dulwich license* 15b⟩≡

```
# Dulwich is dual-licensed under the Apache License, Version 2.0 and the GNU
# General Public License as public by the Free Software Foundation; version 2.0
# or (at your option) any later version. You can redistribute it and/or
# modify it under the terms of either of these two licenses.
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
#
# You should have received a copy of the licenses; if not, see
# <http://www.gnu.org/licenses/> for a copy of the GNU General Public License
# and <http://www.apache.org/licenses/LICENSE-2.0> for a copy of the Apache
# License, Version 2.0.
```

I also sometimes took inspiration from code written by Thomas Gazagnaire for `ocaml-git`[18] (another clone of Git in OCaml but intented to be used as a library in the MirageOS ecosystem[19]).

⟨*copyright ocaml-git* 15c⟩≡ (187b 183b 173b 159b)

```
(*
 * Copyright (c) 2013-2017 Thomas Gazagnaire <thomas@gazagnaire.org>
 *
 * Permission to use, copy, modify, and distribute this software for any
 * purpose with or without fee is hereby granted, provided that the above
 * copyright notice and this permission notice appear in all copies.
 *
 * THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL WARRANTIES
 * WITH REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF
 * MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR
 * ANY SPECIAL, DIRECT, INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
 * WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN
 * ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF
 * OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
 *)
```

The prose and figures are mine and are licensed under the All Rights Reserved License.

[18] https://github.com/mirage/ocaml-git
[19] https://mirage.io/

1.8 Acknowledgments

I would like to acknowledge first the author of Dulwich, Jelmer Vernooij, who wrote in some sense most of this book. Indeed, I used mainly the code of Dulwich to write the code of `ocamlgit`. Dulwich provided an easier path towards understanding the concepts and implementation of Git.

I would like also to thank Linus Torvalds, the original author of Git, for designing this simple but powerful VCS. Many of his design decisions seem obvious in retrospect but they are not: most VCSs chose different designs (for the storage, file permissions, for the way to handle renames, etc.) which in the end are more complicated and usually less efficient than in Git.

Chapter 2

Overview

Before showing the source code of `ocamlgit` in the following chapters, I first give an overview in this chapter of the general principles of a VCS. I also quickly describe the command-line interface of `ocamlgit`, the format of a Git repository, and I use a simple terminal session containing `ocamlgit` commands to illustrate its major features. Finally, I define terms, explain how the code is organized, and more generally give the background necessary to understand the code I will show later.

2.1 Version control system principles

The main requirements for a modern VCS are to be able to store past versions of a set of files, to track the changes to those files, and to help multiple developers to collaborate with each other on those files by supporting concurrent and parallel development (I will explain soon the differences between those two kinds of development). The next sections will each give more details about those requirements. I will also try to present the major techniques used by popular VCSs to satisfy those requirements. Finally, I will discuss two different architectures for VCSs: centralized and distributed.

2.1.1 Storing past versions

The first requirement of a VCS is to store past versions of a file, hence the 'V' in VCS. However, there is also an 'S' in VCS, which means it must be done in a systematic way. Indeed, one way to understand the usefulness of a VCS is to see how people are struggling when they are managing documents that evolves (e.g., source code, LaTeX files, notes) without a VCS. In that situation, people often uses `cp` to *backup* files, but it is inefficient space-wise and error-prone.

The repository

Working copy

Delta

Compression

2.1.2 Tracking changes

The commit

Version identifier

Change granularity

`diff`

`patch`

2.1.3 Concurrent development

Locks

Merging

Merge conflicts

2.1.4 Parallel development

Branch

Merging

A direct acyclic graph of versions

2.1.5 Centralized versus distributed

2.2 `git` command-line interface

I just described the general principles of a VCS, and illustrated some of those principles with examples from RCS, CVS, or Git. I will now focus exclusively on Git and give more details about the interface of its command-line program: `git`. As I mentioned in Section 1.4, the command-line interface of `ocamlgit` is almost identical to the one of the program `git`. This is why I will often use `git` as an alias for `ocamlgit` in the rest of this document.

The command-line interface of `git` (and `ocamlgit`) is pretty simple:

```
1 $ git
2 usage: git <init|add|rm|commit|branch|checkout|reset|...> [options]
3 $ git init
4 Initialized empty Git repository in /tests/hello/.git/
5 $ git add hello.txt
6 $ git commit -m "first commit" --author "pad <todo@todo>"
```

`git` takes first a *command* as its first argument and then options and extra arguments specific to this command. For example, the command `init` at Line 3 does not need any extra arguments, but `add` at Line 5 requires at least the name of a file or directory, and `commit` at Line 6 usually requires command-line flags. I will gradually describe the different `git` commands, their options, and implementations in the following chapters.

2.3 Git concepts and data structures

To understand how Git is implemented, and to some degree to learn how to use Git effectively, you need to know the few concepts underlying Git. Those concepts are the *object store*, the *reference*, and the *staging area*. They correspond also to `ocamlgit`'s main data structures, which I will describe fully in Chapter 3. The following sections will just give an overview of those data structures.

2.3.1 Object store

To really understand Git, you need to realize that at its core, Git is a simple *content-addressable storage system*. Given the *hash* of a content (Git uses SHA1 hashes, as explained below), Git can retrieve back quickly the full corresponding content. Git internally is simply a *persistent hash table*, also known as a *key/value store*.

In this store, Git manipulates mainly three kinds of values, known as *objects* in Git terminology (hence the term *object store*):

- A *blob*: to represent the content of a file at a specific time.
- A *tree*: to assign names to blobs or other subtrees
- A *commit*: to associate to a specific toplevel tree a message, an author, a date, and zero or more *parent* commits

You will see another kind of objects in Section 18.1 with the *tag*. In addition to those objects, Git maintains also *references* to specific commits and an *index* of specific blobs, as explained in the next sections.

Regarding the keys of the store, Git uses the SHA1 algorithm [sha93, EJ01] to compute the hash of an object. This algorithm associates an almost unique number of 160 bits (which amounts to 20 bytes, or 40 digits in hexadecimal format) to any content of arbitrary length. Section 16.1 presents the code of this algorithm and gives more information on SHA1. For example, given the content `Hello Git\n`, the SHA1 algorithm will return the number `9f4d96d5b00d98959ea9960f069585ce42b1349a` in hexadecimal format. Here the hashing is not very interesting because the hash is bigger than the original content, but in practice most files under a VCS are far bigger than 20 bytes.

SHA1 is a complex cryptographic hash function. It is outside the scope of this book to explain how and why it works, but what is important for Git is that SHA1 is an almost perfect hash function: given two different content, there is an almost zero probability that SHA1 will return the same hash number. Such an event is called a *collision*, and in practice it should never happen. Thanks to this almost perfect hash function, Git can identify any content of any length with just a 20 bytes number. In Git, the SHA1 of a file is similar to the *inode* of a file in a filesystem (see the KERNEL book [Pad14] for more information on filesystems and inodes)[1], except it references a specific version of a file. In the tree object, which associates names to blobs, Git uses the SHA1 of a blob to identify this blob. In the same way, in the commit object, Git uses the SHA1 of a tree to identify the toplevel tree a commit refers to.

Figure 2.1 presents the Git objects (and their relationships) of the Git repository created in Section 1.4 after the addition of another commit. The first commit, which added `hello.txt`, is at the top left of Figure 2.1. The second commit, below, added the two files `foo.txt` and `bar.txt` under a directory `dir1/`, and also modified `hello.txt`. I will describe soon in Section 2.4 how Git stores those objects on the disk.

Commit objects are linked together, as shown in Figure 2.1 with the second commit linked to the first commit through the `parents` field of the object. Those links represent the history of the repository, also known as its *log*. A commit can have multiple parents in the case of a *merge* (I will explain in Section 11.1 how to merge branches in Git). Thus, the log forms a *graph*, or more specifically a *direct acyclic graph* (DAG). Figure 2.2 shows such a graph for an additional series of commits following the first two commits of Figure 2.1. In this example, a developer *forked* first an experimental branch called `branch1` (I will explain briefly how to create a

[1] The author of Git, Linus Torvalds, is also the author of Linux, which explains why he chose a design inspired by filesystems.

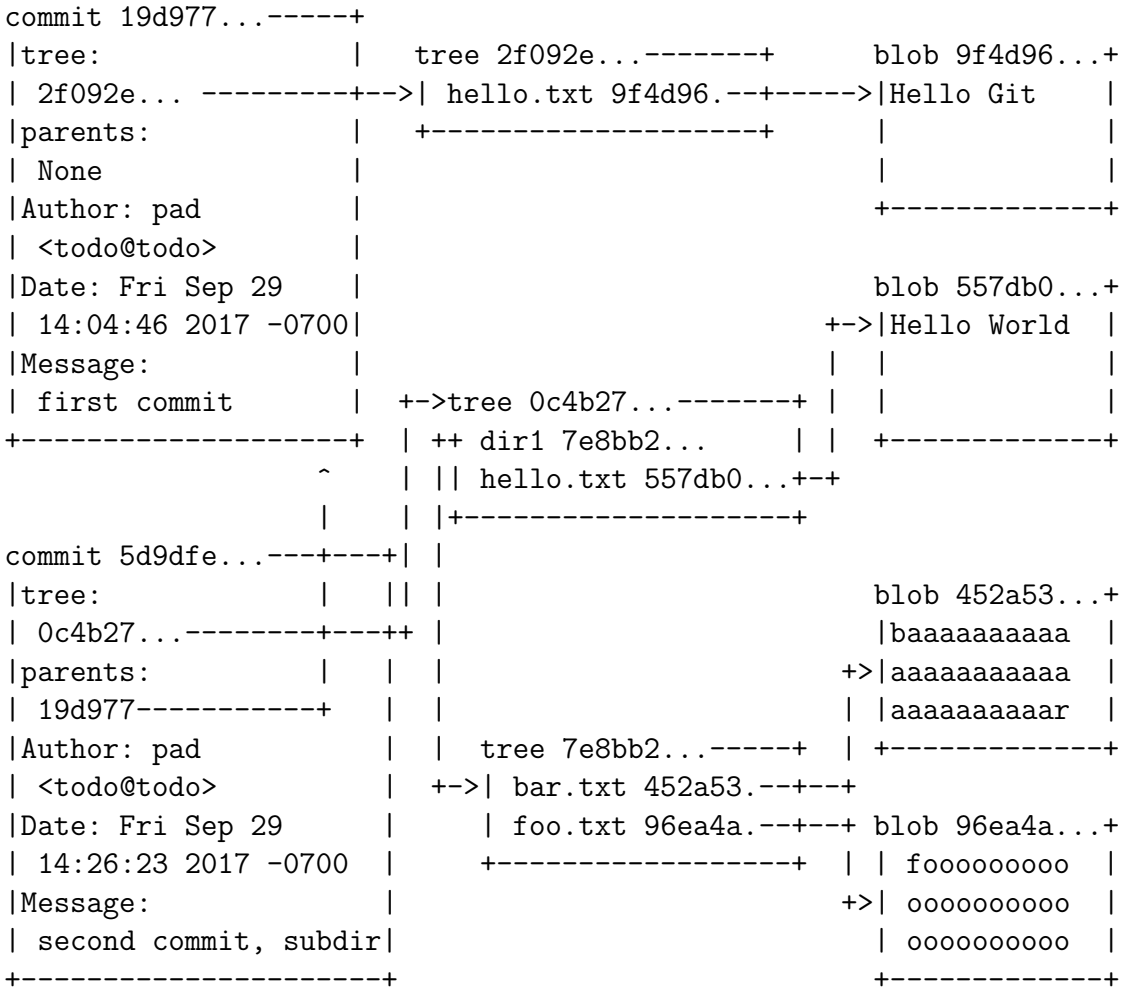

Figure 2.1: Git objects relationships for `hello/.git` after two commits.

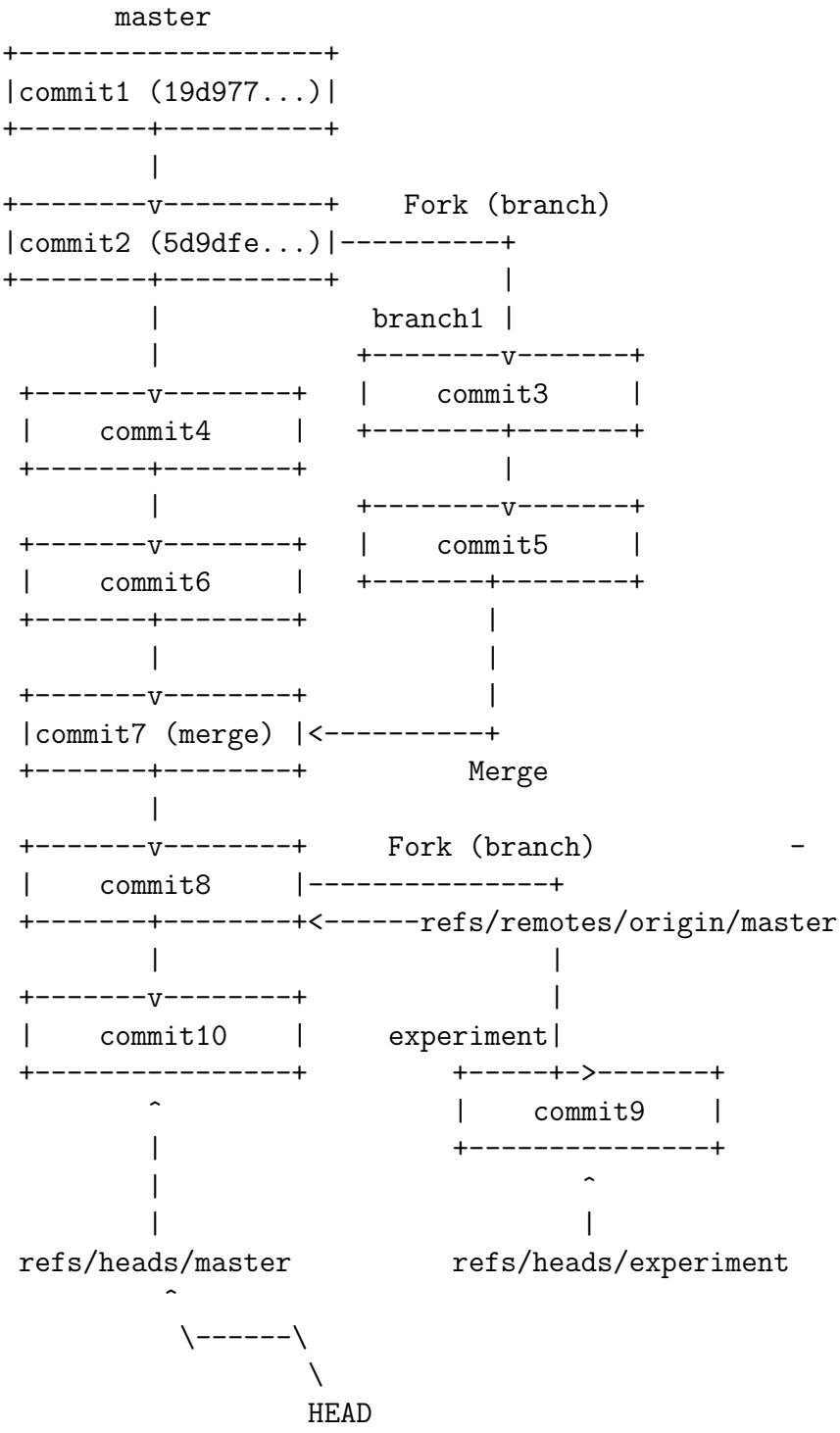

Figure 2.2: Commit graph and references.

branch in Section 2.5.4 and fully in Section 10.2) following the second commit of Figure 2.1. At the same time, development on the main branch, called the *master branch* in Git terminology, continued and saw two commits: `commit4` and `commit6`. The experimental branch got *merged* later at `commit7` (I will explain in Section 11.1 how to merge branches in Git). This is why this commit has two parents. Later on, a developer created another branch called `experiment` that remained active and did not get merged yet; it follows in parallel the development of the master branch.

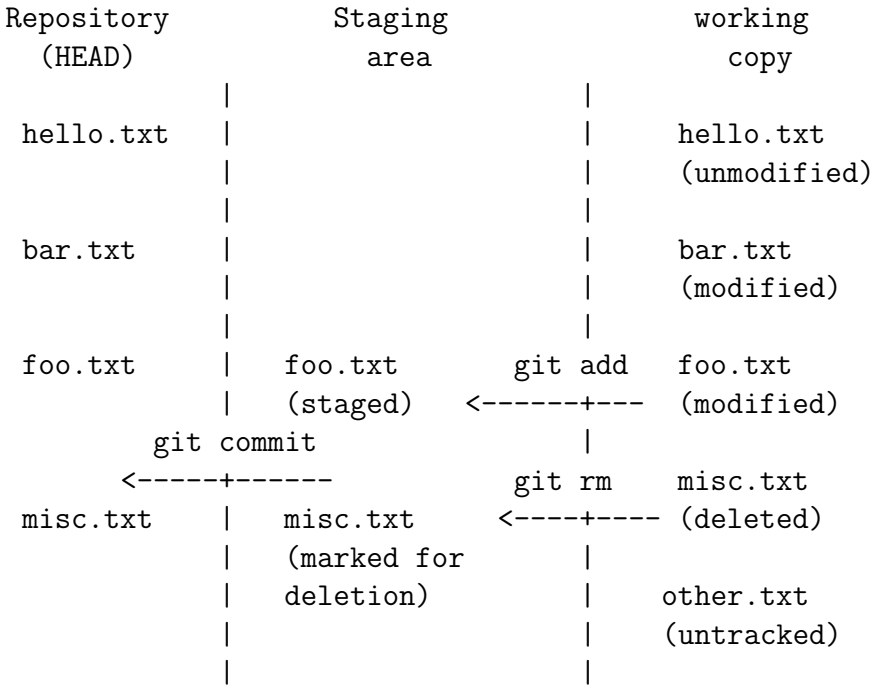

Figure 2.3: Git areas.

2.3.2 References

As I mentionned briefly in the previous section, Git maintains, in addition to the object store, *references* to specific commit objects. Those references have a *name* and a *content*. For example, in Figure 2.2, the reference named `refs/heads/experiment` points to the last commit of the `experiment` branch, and its content is the SHA1 of the `commit9` object. I will explain in Section 2.4 how Git stores those references on the disk.

Git maintains also a special reference called the *HEAD* which points to the last commit of the *current branch* (see the bottom of Figure 2.2). Its content is usually not a SHA1 but the name of another reference. By default, `HEAD` points to the master branch, as shown at the bottom of Figure 2.2, and so its content is the string `ref: refs/heads/master` (and `refs/heads/master` contains the SHA1 of the last commit of the master branch). When you switch branch (I will explain how to switch branch briefly in Section 2.5.4 and fully in Section 10.4), the content of the `HEAD` will change.

A reference pointing to the last commit of a branch is called a *head* in Git terminology and starts with `refs/heads/`. In Figure 2.2, those heads are `refs/heads/experiment` and `refs/heads/master`.

Git maintains a final set of special references called the *remotes*, for example `refs/remotes/origin/master` pointing to `commit8` in Figure 2.2. Those references are used when people are collaborating with each other and I will explain them in Chapter 14.

2.3.3 Staging area

The last important concept of Git is the *staging area*. Git operates mainly on three different areas, as shown at the top of Figure 2.3:

1. The *working copy* contains the current state of the files
2. The *repository* contains all past versions of all the files including the state of the files at the last commit
3. The *staging area* contains what the user wants to commit next

After you modify a set of files in your working copy, for example, `bar.txt`, `foo.txt`, and `misc.txt` at the right of Figure 2.3, you need to indicate which of those files and modifications should be part of the next commit. To do so, you need to *mark* those files by using the command `git add` (or `git rm` if you want to mark for deletion a file).

Once you marked all the relevant files, you can use the command `git commit` to commit the modifications to the repository. For example, in Figure 2.3, only the modifications to `foo.txt` and the deletion of `misc.txt` will be part of the next commit, and not the modifications to `bar.txt`.

Once you add a file in a repository, the file is said to be *tracked* in Git terminology. You can sometimes avoid manipulating explicitely the staging area by using instead the command `git commit -a` to automatically commit all the modifications to the tracked files. However, the staging area and `git add` give more flexibility to the user by allowing to split a set of modifications in multiple commits. For example, in Figure 2.3, the modification to `bar.txt` can be put in another commit.

Internally, Git uses a data structure called the *index* (also known as the *directory cache*) to manage the staging area. I will explain later in Section 3.5 why this data structure is called an index.

2.4 Repository format: `.git/`

Now that you know the main concepts and data structures underlying Git, it will be easy to understand the format of the `.git/` directory. Here is the sligtly edited output of the UNIX command `tree`, which recursively explores a directory and displays its content as a tree [2]. The command is applied here to the `/tests/hello/.git/` directory after the last command in Section 1.4, when the repository contained just one commit:

```
1 $ pwd
/tests/hello
2 $ tree -F .git/
.git/
|-- HEAD
...
|-- index
|-- objects/
|   |-- 2f/
|   |   `-- 092e9cadfc1eb4a6d2febfddb941f4c1fe6fd6
|   |-- 19/
|   |   `-- d977a48d1d7b7ae8d520dd66190702dd4fb5bc
|   |-- 9f/
|   |   `-- 4d96d5b00d98959ea9960f069585ce42b1349a
......
`-- refs/
    |-- heads/
    |   `-- master
.....
```

The objects of the key/value store are simply stored in separate files under `.git/objects/`. The hexadecimal digits of the SHA1 of the object are split in two parts: the first two digits are used for the directory name containing the object and the remaining digits for the filename of the object. For example, the file `.git/objects/9f/4d96d5b00d98959ea9960f069585ce42b1349a` contains the blob of `Hello Git\n` (see the blob and its SHA1 at the top right of Figure 2.1).

[2] See http://mama.indstate.edu/users/ice/tree/ for more information about the `tree` program. The `-F` option used in the terminal session above is to add the special mark `'/'` at the end of directory names, to help differentiate regular files from subdirectories.

Git uses the first two digits of the SHA1 to classify objects in separate folders to avoid having too many files in the same directory. Indeed, this would slow down filesystem operations such as opening a file referenced by a specific SHA1 key, a recurring operation under Git (see the KERNEL book [Pad14] for more information on the performance of filesystem operations).

The references are also stored simply in separate files. Git uses the name of the reference as the path to a file under `.git/refs/` (e.g., `.git/refs/heads/master` in the example above).

Finally, the index and the HEAD are stored directly under `.git/` in separate files.

2.5 hello/.git/

In this section, I will give a short tutorial on how to use Git. I will explain the main commands of `git` by using a terminal session starting from an empty `hello/` directory. I will continue the series of commands I introduced in Section 1.4. I will also describe the semantics of those commands by showing their effects on the Git metadata stored under the `hello/.git/` subdirectory (by using mainly the output of the `tree` command, as in the previous section). This will hopefully help you to understand the code of those commands I will present later in the following chapters.

2.5.1 Creating a repository

Here are the commands to create a fresh new repository:

```
1 $ cd /tests/
2 $ mkdir hello
3 $ cd hello/
4 $ git init
Initialized empty Git repository in /tests/hello/.git/
5 $ tree -F .git/
.git/
|-- HEAD
...
|-- objects/
`-- refs/
    |-- heads/
6 $ cat .git/HEAD
ref: refs/heads/master
```

The `init` command creates just the directory structure under `.git/`, without any objects or references in it (the `.git/objects/` and `.git/refs/heads` are empty directories above). There is also not yet an index file. There is a `HEAD` file, but its contents, shown with command 6 above references an head that does not exist yet under `.git/refs/heads/`.

2.5.2 Staging a diff

Here is the command to add a new file (or a new version of a file) to the repository:

```
1 $ echo "Hello Git" > hello.txt
2 $ git add hello.txt
3 $ tree -F .git/
.git/
|-- HEAD
```

```
...
|-- index
...
|-- objects/
|   |-- 9f/
|   |   `-- 4d96d5b00d98959ea9960f069585ce42b1349a
...
`-- refs/
    |-- heads/
4 $ cat .git/objects/9f/4d96d5b00d98959ea9960f069585ce42b1349a
x^K\312\311OR04`\360H\315\311\311Wp\317,\341^B^@4\201^Ec
```

Adding the first file in a repository has two effects:

1. The creation of a new blob object (here in `.git/objects/9f/4d96...`) containing essentially the compressed form of the content of the added file (hence the cryptic output of the `cat` command above). I will fully explain in Chapter 4 the format of a blob on the disk.

2. The creation of the index file. Again, I will describe precisely the format of the index file in Chapter 4, but the index essentially associates to every tracked filenames by the repository the SHA1 of the blob *currently* corresponding to the filename. In the example above, the index will associate to the filename `hello.txt` the SHA1 `9f4d96...`.

You must use the command `add` to add a new file to the repository, when the file was not yet tracked by the repository. However, you must also use the command `add` to stage the modifications on a tracked file as in Figure 2.3. In that case, the `add` command also creates a new blob with the current content of the file. It also updates the index so the filename of the added file now points to the SHA1 of the new blob.

Note that using `git add` on an unmodified file will have no effect. Indeed, Git will compute the SHA1 of the current content, which will be identical to the SHA1 of an existing blob. Updating the index will have also no effect because the filename will reference the same SHA1.

2.5.3 Committing a diff

Here is the command to commit to the repository what was previously staged:

```
1 $ git commit -m "first commit" --author "pad <todo@todo>"
2 $ tree -F .git/
.git/
|-- HEAD
|-- index
|-- objects/
|   |-- 2f/
|   |   `-- 092e9cadfc1eb4a6d2febfddb941f4c1fe6fd6
|   |-- 19/
|   |   `-- d977a48d1d7b7ae8d520dd66190702dd4fb5bc
|   |-- 9f/
|   |   `-- 4d96d5b00d98959ea9960f069585ce42b1349a
`-- refs/
    |-- heads/
    |   `-- master
3 $ cat .git/refs/heads/master
19d977a48d1d7b7ae8d520dd66190702dd4fb5bc
```

The `commit` command has three effects:

1. The creation of a new tree object, in the example above `.git/objects/2f/092e...` (see Figure 2.1 for the content of this tree). This tree derives from the content of the index. Because the repository contains only one directory (the toplevel root containing just `hello.txt`), `git commit` will create just one tree object, but in general when a project has many directories `git commit` may create many new tree objects.

2. The creation of a new commit object, here `.git/objects/19/d977...`, referencing the newly created tree object (again, see Figure 2.1 for the content of this commit and its relationship to the tree object).

3. The update of the content of the head of the current branch (`refs/heads/master` according to the content of `.git/HEAD`) to contain the SHA1 of the newly created commit object (here `19d977...`), as shown by command 3 above.

Note that it can be tedious to specify each time on the command-line your name and email through the `--author` flag of `git commit`. Git can also leverage a configuration file containing such information in a `.gitconfig` file in your home directory as explained in Section 17.3.

2.5.4 Managing branches

Here is the command to create a new branch:

```
1 $ git branch experiment
2 $ tree -F .git/
.git/
|-- HEAD
...
‘-- refs/
    |-- heads/
    |   |-- experiment
    |   ‘-- master
...
3 $ cat .git/HEAD
ref: refs/heads/master
4 $ cat .git/refs/heads/master
19d977a48d1d7b7ae8d520dd66190702dd4fb5bc
5 $ cat .git/refs/heads/experiment
19d977a48d1d7b7ae8d520dd66190702dd4fb5bc
```

As you can see, creating a new branch is an extremely cheap operation under Git. It consists just in adding a file under `.git/refs/heads/` with the name of the new branch (here `.git/refs/heads/experiment`), and the SHA1 commit of the current branch for its content (here `19d977...`). However, creating a branch does not switch to this branch as shown by command 3 above.

Here is the command to switch to the new branch:

```
6 $ git checkout experiment
Switched to branch ’experiment’
7 $ cat .git/HEAD
ref: refs/heads/experiment
```

Switching to a branch is usually a more costly operation. `git checkout` modifies `HEAD` (as shown by command 7 above), which is fast, but it also recomputes the index from the tree object that is referenced in the commit of the head of the new branch, as well as from all its subtrees (see Section 10.4.1 for the full description of this operation). It also updates all the files so that they contain the content of the blobs referenced in the index. The more the new branch differs from the current branch, the more costly switching to this new branch will be.

2.5.5 Inspecting objects

Git provides a few commands to query the repository. Those commands are useful to understand the internal structures of Git. They are also useful for doing some archeology on the history of a project, as explained in Section 12.7. The first of those commands, `show`, shows the content of a Git object. `git show` takes the SHA1 of an object as a parameter (in hexadecimal format) and pretty prints its content in a readable format. Here are a few examples of this command:

```
1 $ tree -F .git/
...
|-- objects/
|   |-- 2f/
|   |   `-- 092e9cadfc1eb4a6d2febfddb941f4c1fe6fd6
|   |-- 19/
|   |   `-- d977a48d1d7b7ae8d520dd66190702dd4fb5bc
|   |-- 9f/
|   |   `-- 4d96d5b00d98959ea9960f069585ce42b1349a
...
2 $ git show 9f4d96d5b00d98959ea9960f069585ce42b1349a
Hello Git
3 $ git show 2f092e9cadfc1eb4a6d2febfddb941f4c1fe6fd6
tree 2f092e9cadfc1eb4a6d2febfddb941f4c1fe6fd6

hello.txt
4 $ git show 19d977a48d1d7b7ae8d520dd66190702dd4fb5bc
commit 19d977a48d1d7b7ae8d520dd66190702dd4fb5bc
Author: pad  <todo@todo>
Date:   Fri Sep 29 14:04:46 2017 -0700

    first commit

diff --git a/hello.txt b/hello.txt
new file mode 100644
index 0000000..9f4d96d
--- /dev/null
+++ b/hello.txt
@@ -0,0 +1 @@
+Hello Git
```

The command `show` mainly opens the appropriate file under `.git/objects/`, decompresses the file, and finally displays its content.

Another important query command is `log` to see the full history of a repository. Here is the output of this command on the repository we used this far:

```
1 $ git log
commit 19d977a48d1d7b7ae8d520dd66190702dd4fb5bc
Author: pad  <todo@todo>
Date:   Fri Sep 29 14:04:46 2017 -0700

    first commit
```

The command `log` mainly opens `.git/HEAD` to get a reference to the last commit. Then, it goes through the parent links of the commit object to recursively explore all the linked commit objects (stored again under `.git/objects/`).

2.5.6 Other commands

There are a few other important Git commands: `git merge`, `git status`, `git diff`, `git clone`, `git pull`, or `git push`, but I will introduce them later in this document.

2.6 Code organization

Table 2.1 presents short descriptions of the source files used by `ocamlgit` and the corresponding chapters in this document in which the code contained in the file is primarily discussed.

The most important files are `repository.ml`[180], which contains the main API to manipulate a repository, and the `cmd_xxx.ml` files implementing each a Git command (e.g., `cmd_init.ml`[155a] for `git init`).

2.7 Software architecture

Figure 2.4 describes the main modules of `ocamlgit` and their main dependencies. Those dependencies correspond roughly also to the main control flow of `ocamlgit`. This flow starts at the top of Figure 2.4 with the execution of `Main.main()`[56b] when the program `ocamlgit` starts. This function looks whether the first argument of `ocamlgit` is one of the command listed in the global `Cmds.main_commands`[55b]. If it is, then `Main.main()` dispatches the appropriate command, for example, `Cmd_add.cmd`[62a] if the first argument to `ocamlgit` was `add`. Each command usually processes the remaining command-line arguments to `ocamlgit` and uses functions from the API provided by the `Repository` module. This API contains functions to open a repository (`Repository.open_()`[61b]), read an object (`Repository.read_obj()`[41b]), add an object (`Repository.add_obj()`[49b]), modify a reference (`Repository.set_ref()`[100c]), modify the index (`Repository.add_in_index()`[63b]), etc. Internally, the `Repository` module relies on the `Objects`, `Refs`, and `Index` modules to read and write files under the `.git/` directory (e.g., `Index.read()`[53e] and `Index.write()`[47d] to respectively read and write the index file in `.git/index`). Those modules in turn rely on algorithms to compute SHA1 hashes (`Sha1.sha1()`[110b]), to list the differences between two files (`Diffs.diff()`[128c]), to merge two files coming from a common base file (`Diff3.diff3()`[131c]), or to compress (`Zip.deflate()`[126b]) and decompress (`Unzip.inflate()`[185]) data.

2.8 Book structure

You now have enough background to understand the source code of `ocamlgit`. The rest of the book is organized as follows. I will start by describing the core data structures of `ocamlgit` in Chapter 3. Chapter 4 and Chapter 5 contains respectively the code to read and write those data structures on the disk in files under `.git/`. Then, I will switch to a top-down approach, starting with Chapter 6 with the description of `Main.main()`[56b] and the dispatch of the Git commands. In Chapter 7 I will describe the code for the `init` command, which initializes the `.git/` directory. The following five chapters will describe the commands of `ocamlgit` that cover most of the

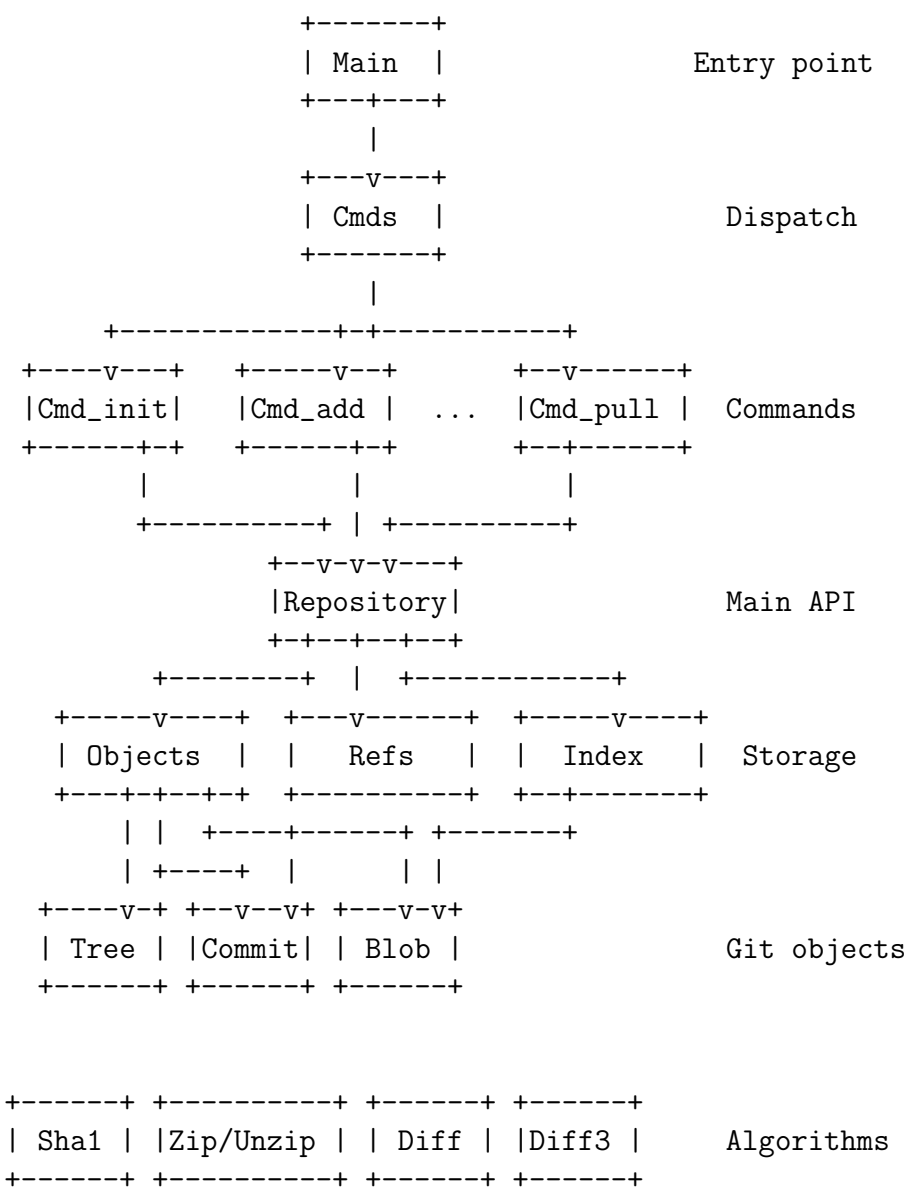

Figure 2.4: Main module dependencies of `ocamlgit`.

Function	Chapter	Files
SHA1 binary and hexadecimal hashes	3	`sha1.mli hexsha.ml`
repository type and main API	3	`repository.ml`
objects	3	`objects.ml blob.ml tree.ml commit.ml user.ml`
references	3	`refs.ml`
staging area	3	`index.ml`
reading from a repository	4	`decompression.ml`
writing to a repository	5	`compression.ml`
list of commands	6	`cmd_.ml cmds.ml`
entry point and command dispatcher	6	`main.ml`
getting help	6	`cmd_help.ml`
creating a repository	7	`cmd_init.ml`
staging a diff	8	`cmd_add.ml cmd_rm.ml cmd_mv.ml`
committing a diff	9	`cmd_commit.ml`
manipulating branches	10	`cmd_branch.ml cmd_checkout.ml cmd_reset.ml`
merging multiple branches	11	`cmd_merge.ml`
inspecting objects	12	`cmd_show.ml`
tree and file changes	12	`change.ml diff.ml`
showing differences	12	`cmd_diff.ml changes.ml diff_unified.ml`
commit history	12	`cmd_log.ml`
file status	12	`cmd_status.ml`
packing objects and delta compression	13	`pack.ml`
exchanging commits	14	`cmd_pull.ml cmd_push.ml`
local exchange	14	`client_local.ml`
cloning a repository	14	`cmd_clone.ml`
client/server architecture	15	`client.ml server.ml`
networking clients	15	`clients.ml`
`git://` protocol	15	`client_git.ml`
core algorithms	16	`sha1.ml zip.ml unzip.ml diff_basic.ml diff_myers.ml d`
advanced features	17	`config.ml ignore.ml`
advanced commands	18	`tag.ml`
advanced networking	19	
dumpers and debugging support	A	`cmd_dump.ml dump.ml cmd_test.ml`
binary IO utilities	D	`IO_.ml`
Total		

Table 2.1: Chapters and source files of `ocamlgit`.

use-cases for a single developer: Chapter 8 will present the code to add files in the staging area, Chapter 9 the code to commit what was staged, Chapter 10 the code to create and switch branches, Chapter 11 the code to merge branches, and finally Chapter 12 the code to query the repository. Chapter 13 presents the code to pack and compress objects, which is an important optimization. Then I will present the commands for collaborating with other developers: Chapter 14 presents the code to exchange commits between repositories, and Chapter 15

the code to exchange those commits through the network. The code in the previous chapters rely on general algorithms that are useful not only for Git but also for other programs. I will present the code of those general algorithms in Chapter 16 (e.g., the algorithm to compute the SHA1 of any string, or the algorithm to compute the differences between two files). Then, I will present advanced functionalities of Git that I did not present before to simplify the explanations. Starting with Chapter 17, I will presents advanced features, for instance, the `.gitconfig` configuration file. In Chapter 18 I will present advanced commands, for instance, `git tags`. Finally in Chapter 19 I will present advanced networking options, for instance, a Git client and Git server using the `http://` protocol. Chapter 21 concludes and gives pointers to other books in the Principia Softwarica series.

Some appendices present the code of non-functional properties: code to help debug `ocamlgit` itself in Appendix A. Finally, Appendix D contains the code of generic utility functions used by `ocamlgit` but which are not specific to `ocamlgit`.

Chapter 3

Core Data Structures

Show me your code and conceal your data structures, and I shall continue to be mystified. Show me your data structures, and I won't usually need your code; it'll be obvious.

Fred Brooks

I mentioned before in Section 2.3 the core data structures of Git: the object store, the reference, and the index. This chapter contains three sections introducing the types of those three data structures. Those data structures contain SHA1 hashes, so I will present first the types of the different kinds of SHA1 hashes (binary and hexadecimal).

3.1 Secure Hash Algorithm (SHA1) hashes

Git relies heavily on SHA1 hashes to name and address Git objects. Git uses two different formats for those hashes: binary hashes (mainly for storage on the disk) and hexadecimal hashes (mainly to interact with the user) as explained in the following sections.

3.1.1 Binary hashes

As I mentioned in Section 2.3.1, an SHA1 hash is a 160 bits number. Such a number can be encoded as a sequence of 20 bytes.

⟨*type* `Sha1.t` 32a⟩≡ (182)
```
(* a 20 bytes number (really a string of length 20) *)
type t = string
```

The `sha1()` function returns a 20 bytes number for any content of arbitrary length.

⟨*signature* `Sha1.sha1` 32b⟩≡ (182a)
```
(* computes SHA1 of a series of bytes *)
val sha1: string -> t
```

I will present the code of `Sha1.sha1()`[110b] later in Section 16.1.

OCaml types are not powerful enough to express that a binary SHA1 hash is a sequence of precisely 20 bytes[1]. The function below expresses this additional constraint:

⟨*signature* `Sha1.is_sha` 32c⟩≡ (182a)
```
val is_sha: t -> bool
```

⟨*function* `Sha1.is_sha` 32d⟩≡ (182b)
```
let is_sha (x : t) : bool =
  String.length x = 20
```

[1] We could use a tuple with 20 character elements, but that would be tedious to use.

This function will be used mostly as a form of defensive programming in asserts.

3.1.2 Hexadecimal hashes

A 160 bits number can also be represented as 40 characters in hexadecimal (two hexadecimal characters per byte, e.g., `"ff"` for 255).

⟨*type* `Hexsha.t` 33a⟩≡ (172)
```
(* a 40 characters string, e.g. "d670460b4b4aece5915caf5c68d12f560a9fe3e4" *)
type t = string
```

Again, OCaml types are not powerful enough to fully express the format of an hexadecimal SHA1 hash, so the function below expresses the remaining constraints:

⟨*signature* `Hexsha.is_hexsha` 33b⟩≡ (172a)
```
val is_hexsha: t -> bool
```

⟨*function* `Hexsha.is_hexsha` 33c⟩≡ (172b)
```
let is_hexsha x =
 String.length x = 40 && x =~ "^[0-9a-fA-F]+$"
```

3.1.3 Conversion functions

Both formats of SHA1 hashes are useful and we will need many times to convert from one format to the other. The first conversion function transforms an hexadecimal hash into a binary hash.

⟨*signature* `Hexsha.to_sha` 33d⟩≡ (172a)
```
val to_sha: t -> Sha1.t
```

⟨*function* `Hexsha.to_sha` 33e⟩≡ (172b)
```
let to_sha s =
  assert (is_hexsha s);
  let n = String.length s in
  let buf = Bytes.create (n/2) in
  ⟨Hexsha.to_sha() fill buf 33f⟩
  Bytes.to_string buf
```
Uses `Hexsha.is_hexsha()`.

The function `to_sha()` relies on the nested function below to iterate over all the pairs of characters in the hexadecimal hash and to generate one byte from each pair:

⟨`Hexsha.to_sha()` *fill* `buf` 33f⟩≡ (33e)
```
let rec aux i =
  if i >= n
  then ()
  else begin
    ⟨hexsha.to_sha() double sanity check range i 34b⟩
    Bytes.set buf (i/2) (to_byte s.[i] s.[i+1]);
    aux (i+2)
  end
in
aux 0;
```

⟨*function* `Hexsha.to_char` 34a⟩≡ (172b)
```
let to_byte hex1 hex2 =
  let code hex =
    match hex with
    | '0'..'9' -> Char.code hex - Char.code '0'
    | 'A'..'F' -> Char.code hex - (Char.code 'A' + 10)
    | 'a'..'f' -> Char.code hex - (Char.code 'a' + 10)
    | _ ->
      raise (Invalid_argument
              (spf "Hex.to_byte: %d is an invalid hexadecimal digit" (Char.code hex)))
  in
  Char.chr (((code hex1) lsl 4) + (code hex2))
```
Uses `Common.spf()`.

⟨`hexsha.to_sha()` *double sanity check range i* 34b⟩≡ (33f)
```
if i+1 >= n
then raise (Invalid_argument "hex conversion: invalid hex string");
```

The other conversion function goes the other way around, from a binary hash to an hexadecimal hash:

⟨*signature* `Hexsha.of_sha` 34c⟩≡ (172a)
```
val of_sha: Sha1.t -> t
```

⟨*function* `Hexsha.of_sha` 34d⟩≡ (172b)
```
let of_sha s =
  assert (Sha1.is_sha s);
  let n = String.length s in
  let buf = Bytes.create (n * 2) in
  ⟨Hexsha.of_sha() fill buf 34e⟩
  Bytes.to_string buf
```
Uses `Sha1.is_sha()` 32d.

This time, each byte from `s` generates two characters:

⟨`Hexsha.of_sha()` *fill* `buf` 34e⟩≡ (34d)
```
for i = 0 to n - 1 do
  Bytes.set buf (i * 2) hexa1.[Char.code (s.[i])];
  Bytes.set buf ((i * 2) + 1) hexa2.[Char.code (s.[i])];
done;
```
Uses `Hexsha.hexa1` and `Hexsha.hexa2`.

The loop above relies on two constants containing 256 elements mapping a byte to one of the two hexadecimal characters:

⟨*constant* `Hexsha.hexa1` 34f⟩≡ (172b)
```
and hexa1 =
  "0000000000000000111111111111111122222222222222223333333333333333\
   4444444444444444555555555555555566666666666666667777777777777777\
   8888888888888888999999999999999aaaaaaaaaaaaaaabbbbbbbbbbbbbbbb\
   ccccccccccccccccddddddddddddddddeeeeeeeeeeeeeeeeffffffffffffffff"
```

⟨*constant* `Hexsha.hexa2` 34g⟩≡ (172b)
```
and hexa2 =
  "0123456789abcdef0123456789abcdef0123456789abcdef0123456789abcdef\
   0123456789abcdef0123456789abcdef0123456789abcdef0123456789abcdef\
   0123456789abcdef0123456789abcdef0123456789abcdef0123456789abcdef\
   0123456789abcdef0123456789abcdef0123456789abcdef0123456789abcdef"
```

3.2 Repository.t

In the following chapters, many functions will require to know where is located the repository they need to operate on. Those functions will need to access files under the `.git/` subdirectory or to access the working files of a repository. The type below conveniently packs all the necessary information to locate files on the disk:

⟨*type* `Repository.t` 35a⟩≡ (180 179)
```
type t = {
  worktree: Fpath.t;
  dotgit: Fpath.t; (* usually <worktree>/.git *)

  ⟨Repository.t index field 40b⟩
}
```

Given a repository `r` and an hexadecimal SHA1 hash `hexsha`, it is easy to locate a Git object on the disk thanks to the following function:

⟨*function* `Repository.hexsha_to_filename` 35b⟩≡ (180)
```
(* for loose objects *)
let hexsha_to_filename (r : t) (hexsha : string) : Fpath.t =
  let dir = String.sub hexsha 0 2 in
  let file = String.sub hexsha 2 (String.length hexsha - 2) in
  r.dotgit / "objects" / dir / file
```
Uses `Repository./()` and `Repository.t.dotgit` 35a.

The function above relies on a special operator to concatenate filenames:

⟨*constant* `Repository.SlashOperator` 35c⟩≡ (180)

3.3 Objects.t

As I explained in Section 2.3.1, Git uses internally mainly three kinds of objects, as expressed by the following type:

⟨*type* `Objects.t` 35d⟩≡ (177)
```
type t =
  | Blob   of Blob.t
  | Tree   of Tree.t
  | Commit of Commit.t
  ⟨Objects.t cases 134a⟩
```
Uses `Commit.t` 36e.

The following sections will detail `Blob.t`[35e], `Tree.t`[36a], and `Commit.t`[36e]. The following chapter will describe the format of those objects on the disk.

3.3.1 Blob.t

A blob is simply a sequence of bytes.

⟨*type* `Blob.t` 35e⟩≡ (149a 148c)
```
type t = string
```

In the following data structures, some fields will contain SHA1 hashes that must refer to a blob. The type below can be used to express this constraint; this constraint will not be checked statically, but using the type provides a useful comment:

⟨*type* `Blob.hash` 35f⟩≡ (149a 148c)
```
type hash = Sha1.t
```

3.3.2 Tree.t

A tree is an ordered list of entries where each entry associates to a name a Git object through its (binary) SHA1.

⟨*type* Tree.t 36a⟩≡ (183)
```
(* todo: entries must be sorted! and each name must be unique *)
type t = entry list
```

⟨*type* Tree.entry 36b⟩≡ (183)
```
type entry = {
  (* relative to tree, so does not contain any '/', or '.' or '..' *)
  name: string;
  (* Blob.hash or Tree.hash *)
  id: Sha1.t;

  perm: perm;
}
```
Uses Tree.perm.

⟨*type* Tree.hash 36c⟩≡ (183)
```
type hash = Sha1.t
```

Each name refers either to a file (a blob), or a directory (another tree).

Git handles only a few file types and permissions for those entries, as expressed by the type below:

⟨*type* Tree.perm 36d⟩≡ (183)
```
(* similar to Index.mode, but with also a 'Dir' *)
type perm =
  | Normal
  | Dir
  | Exec
  ⟨Tree.perm cases 132a⟩
```

I will present in Chapter 17 a few more file types supported by Git (e.g., symbolic links, submodules).

Note that a tree entry does not specify the used id (`uid`) or group id (`gid`) of a file, or the read-write-execute permissions (`rwx`) of a file (except for the `x` with `Exec` above to denote executable files). Indeed, the repository must abstract away this kind of specific information. Many people can collaborate with each other by working on the same repository, so it would not make sense to favor any user or permission.

Git supports only a simple subset of UNIX file types. It can not store special files such as a socket, a named pipe, or a device file.

3.3.3 Commit.t

The last Git object, the commit, associates to a toplevel tree (again through its SHA1 binary hash) a set of parent commits, an author, a date, and a commit message.

⟨*type* Commit.t 36e⟩≡ (159)
```
type t = {
  tree     : Tree.hash; (* the root *)
  (* first commit has no parent, and merge commits have 2 parents *)
  parents  : hash list;
  (* note that User.t contains a time *)
  author   : User.t;
  message  : string;
  ⟨Commit.t extra fields ??⟩
}
```
Uses User.t 37a.

⟨*type* Commit.hash 36f⟩≡ (159)
```
and hash = Sha1.t
```

The `User.t` type below specifies the name and email of an author, but also the date of the commit and the timezone of the author.

⟨*type* `User.t` 37a⟩≡ (187)
```
type t = {
  name : string;
  email: string;
  date : Int64.t (* seconds *) * timezone_offset;
}
```

⟨*type* `User.tz_offset` 37b⟩≡ (187)
```
type timezone_offset = int (* +/- hours, [-12, +12] *)
```

Git allows to differentiate who was the author of a modification and who actually commited the modification to the repository.

⟨`Commit.t` *extra fields* 37c⟩+≡ (36e) ◁??
```
committer: User.t;
```
Uses `User.t` 37a.

3.4 References

As I explained in Section 2.3.2, references, which are mainly stored under `.git/refs/`, are another fundamental data structure of Git. References have a name (e.g., `refs/heads/master`), and a content (usually an hexadecimal SHA1 hash), as explained in the following sections.

3.4.1 Reference names

⟨*type* `Refs.refname` 37d⟩≡ (178)
```
(* should always start with "refs/", see is_refname() later *)
type refname = string (* e.g. "refs/heads/master" *)
```

The format of a reference name has a few constraints that the type can not easily express:

⟨*signature* `Refs.is_valid_refname` 37e⟩≡ (178a)
```
val is_refname: refname -> bool
```

⟨*function* `Refs.is_valid_refname` 37f⟩≡ (178b)
```
let is_refname str =
  str =~ "^refs/"
  (* todo: git-check-ref-format *)
```

3.4.2 `HEAD`

Most references are stored under `.git/refs/`, but Git maintains also a special reference called the `HEAD` (see Section 2.3.2) to point to the head of the current branch, hence the following type:

⟨*type* `Refs.t` 37g⟩≡ (178)
```
type t =
  | Head
  | Ref of refname
```

⟨*signature* `Refs.string_of_ref` 37h⟩≡ (178a)
```
val string_of_ref: t -> string
```

⟨*function* `Refs.string_of_ref` 38a⟩≡ (178b)
```
let string_of_ref = function
  | Head -> "HEAD"
  | Ref x -> x
```
Uses `Refs.t.Head` 37g and `Refs.t.Ref` 37g.

3.4.3 Reference content

Most references contain the SHA1 hexadecimal hash of a commit, but they can also contain the name of another reference:

⟨*type* `Refs.ref_content` 38b⟩≡ (178)
```
type ref_content =
  (* the final value when follow all the pointers *)
  | Hash of Commit.hash
  (* pointer (may contain sha1 or another pointer again) *)
  | OtherRef of refname
```

By default, `HEAD` points to another reference:

⟨*signature* `Refs.default_head_content` 38c⟩≡ (178a)
```
val default_head_content: ref_content
```

⟨*constant* `Refs.default_head_content` 38d⟩≡ (178b)
```
let default_head_content =
  OtherRef "refs/heads/master"
```
Uses `Refs.ref_content.OtherRef` 38b.

3.4.4 Reference store

Similar to the objects in Section 3.2 with `Repository.hexsha_to_filename()`[35b], the function below computes the path of a reference:

⟨*function* `Repository.ref_to_filename` 38e⟩≡ (180)
```
let ref_to_filename r aref =
  match aref with
  | Refs.Head -> r.dotgit / "HEAD"
  (* less: win32: should actually replace '/' in name *)
  | Refs.Ref name -> r.dotgit / name
```
Uses `Refs.t.Head` 37g, `Refs.t.Ref` 37g, `Repository./()`, and `Repository.t.dotgit` 35a.

3.4.5 Objectish references

Many commands of Git requires a commit as a parameter, for example `git checkout`. It is convenient for the user to be able to specify a commit either through its hexadecimal SHA1, or through the name of a branch, or through a reference, hence the type below:

⟨*type* `Repository.objectish` 38f⟩≡ (180 179)
```
(* todo: handle ^ like HEAD^, so need more complex objectish parser *)
type objectish =
  | ObjByRef of Refs.t
  | ObjByHex of Hexsha.t
  | ObjByBranch of string
  ⟨Repository.objectish cases ??⟩
```
Uses `Refs.t`.

3.5 Index.t

The last important Git data structure is the index stored under `.git/index`.

⟨*function* `Repository.index_to_filename` 39a⟩≡ (180)
```
let index_to_filename r =
  r.dotgit / "index"
```
Uses `Repository./()` and `Repository.t.dotgit` 35a.

An index is a sorted list of entries (similar to `Tree.t`[36a]) where each entry associates to a path an SHA1 binary hash.

⟨*type* `Index.t` 39b⟩≡ (173)
```
(* the entries are sorted (see compare_entries below) *)
type t = entry list
```

⟨*type* `Index.entry` 39c⟩≡ (173)
```
(** The type for a Git index entry. *)
type entry = {
  (* relative path *)
  path  : Fpath.t;
  id    : Blob.hash;

  stats : stat_info;
}
```
Uses `Index.stat_info`.

As opposed to `Tree.entry`[183b], an `Index.entry`[39c] contains a path to a file (e.g., `dir/bar.txt`), not the basename of file or directory. Indeed, the type of the `id` field above refers only to the SHA1 of a blob. Moreover, instead of a small set of file types, an index entry contains the full UNIX stat of a file (including its `uid` and `gid`):

⟨*type* `Index.stat_info` 39d⟩≡ (173)
```
(** The type for file-system stat information. *)
type stat_info = {
  mode : mode;

  ctime: time;
  mtime: time;

  dev  : Int32.t;
  inode: Int32.t;

  uid  : Int32.t;
  gid  : Int32.t;

  size : Int32.t;
}
```
Uses `Index.mode` and `Index.time`.

An index entry is specific to a user. The goal of the index, which was initially called the *directory cache* in the first version of Git, is to access quickly information about a tracked file, for example the timestamp of a file at the moment it was checkout.

⟨*type* `Index.mode` 39e⟩≡ (173)
```
and mode =
  (* no directory here *)
  | Normal
  | Exec
  ⟨Index.mode cases 132b⟩
```

⟨*type* `Index.time` 40a⟩≡ (173)
```
(** The type for a time represented by its [lsb32] and [nsec] parts. *)
and time = {
  lsb32: Int32.t;
  nsec : Int32.t;
}
```

Most Git commands will need information from the index (e.g., `git commit`), or will modify the index (e.g., `git add`), so the index is one of the fields of the `Repository.t`[35a] structure:

⟨`Repository.t` *index field* 40b⟩≡ (35a)
```
mutable index: Index.t;
```

Opening a repository (with `Repository.open_()`[61b]), will read the index from the repository. If there is no index file yet (for example after `git init` and before the first `git add`), then `ocamlgit` uses an empty index thanks to the constant below:

⟨*signature* `Index.empty` 40c⟩≡ (173a)
```
val empty: t
```

⟨*constant* `Index.empty` 40d⟩≡ (173b)
```
let empty = []
```

The function below helps to build an index entry from its subelements:

⟨*signature* `Index.mk_entry` 40e⟩≡ (173a)
```
val mk_entry: Fpath.t -> Sha1.t -> Unix.stats -> entry
```

⟨*function* `Index.mk_entry` 40f⟩≡ (173b)
```
let mk_entry relpath sha stats =
  { path = relpath;
    id = sha;
    stats = stat_info_of_lstats stats;
  }
```
Uses `Index.entry.id` 39c, `Index.entry.stats` 39c, and `Index.stat_info_of_lstats()` 173b.

⟨*signature* `Index.stat_info_of_lstats` 40g⟩≡ (173a)
```
val stat_info_of_lstats: Unix.stats -> stat_info
```

⟨*function* `Index.stat_info_of_lstats` 40h⟩≡ (173b)
```
let stat_info_of_lstats stats =
    { ctime = { lsb32 = Int32.of_float stats.Unix.st_ctime; nsec = 0l };
      mtime = { lsb32 = Int32.of_float stats.Unix.st_mtime; nsec = 0l };
      dev = Int32.of_int stats.Unix.st_dev;
      inode = Int32.of_int stats.Unix.st_ino;
      mode =
        (match stats.Unix.st_kind, stats.Unix.st_perm with
        | Unix.S_REG, p ->
          if p land 0o100 = 0o100
          then Exec
          else Normal
        ⟨Index.stat_info_of_lstats() match kind and perm cases 132c⟩
        | _ -> failwith ("unsupported file type")
        );
      uid = Int32.of_int stats.Unix.st_uid;
      gid = Int32.of_int stats.Unix.st_gid;
      size = Int32.of_int stats.Unix.st_size;
    }
```
Uses `Index.mode.Exec` 39e, `Index.mode.Link` 39e, `Index.mode.Normal` 39e, `Index.stat_info.dev` 39d, `Index.stat_info.gid` 39d, `Index.stat_info.inode` 39d, `Index.stat_info.mode` 39d, `Index.stat_info.mtime` 39d, `Index.stat_info.size` 39d, `Index.stat_info.uid` 39d, `Index.time.lsb32` 40a, and `Index.time.nsec` 40a.

Chapter 4

Reading from a Repository

Now that you know the types of the core data structures and how they are represented in memory, we can look at the format of those data structures on the disk. In this chapter, I will describe the code that reads those data structures from the disk, with the code of `Repository.read_obj()`[41b], `Repository.read_ref()`[45e], and `Repository.read_index()`[47a]. The next chapter will describe the code that writes those data structures on the disk.

4.1 Objects

⟨*signature* `Repository.read_obj` 41a⟩≡ (179)
```
val read_obj: t -> Sha1.t -> Objects.t
```

⟨*function* `Repository.read_obj` 41b⟩≡ (180)
```
let read_obj r h =
  (* todo: look for packed obj *)
  let path = h |> Hexsha.of_sha |> hexsha_to_filename r in
  path |> UChan.with_open_in (fun (ch : Chan.i) ->
    (* less: check read everything from channel? *)
    (* todo: check if sha consistent? *)
    ch.ic |> IO.input_channel |> Compression.decompress |> Objects.read
  )
```
Uses `Common.with_file_in()`, `Compression.decompress()` 42b, `Hexsha.of_sha()`, `IO.input_channel()`, `Objects.read()` 41d, and `Repository.hexsha_to_filename()` 35b.

⟨*signature* `Objects.read` 41c⟩≡ (177b)
```
(* assumes input is in decompressed form *)
val read: IO.input -> t
```

⟨*function* `Objects.read` 41d⟩≡ (177c)
```
let read ch =
  let str = IO_.read_string_and_stop_char ch ' ' in
  let n = IO_.read_int_and_nullbyte ch in
  let raw = IO.really_nread ch n in
  (* less: assert finished ch? use IO.pos_in? *)
  let ch2 = IO.input_bytes raw in
  (* less: just reuse ch so avoid use of intermediate strings? *)
  match str with
  ⟨Objects.read() match str cases 42c⟩
  (* less: assert finished ch2? *)
  | str -> failwith (spf "Objects.read: invalid header: %s" str)
```
Uses `Common.spf()`, `IO.input_bytes()`, `IO.really_nread()`, `IO_.read_int_and_nullbyte()` 146g, and `IO_.read_string_and_stop_cha` 146e.

4.1.1 Decompression

⟨*signature* `Compression.decompress` 42a⟩≡ (160a)
```
val decompress:
  IO.input -> IO.input
```

⟨*function* `Compression.decompress` 42b⟩≡ (160b)
```
let decompress ch =
  Unzip.inflate ch
```

4.1.2 Reading a blob

⟨`Objects.read()` *match str cases* 42c⟩≡ (41d) 42h▷
```
| "blob"   -> Blob   (Blob.read ch2)
```
Uses `Blob.read()` 42e and `Objects.t.Blob` 35d.

⟨*signature* `Blob.read` 42d⟩≡ (148c)
```
(* assumes have already read the 'blob <size>\000' header from unzipped input *)
val read: IO.input -> t
```

⟨*function* `Blob.read` 42e⟩≡ (149a)
```
let read ch =
  IO.read_all ch
```

⟨*signature* `Repository.read_blob` 42f⟩≡ (179)
```
val read_blob: t -> Sha1.t -> Blob.t
```

⟨*function* `Repository.read_blob` 42g⟩≡ (180)
```
let read_blob r h =
  match read_obj r h with
  | Objects.Blob x -> x
  | _ -> failwith "read_blob: was expecting a blob"
```
Uses `Objects.t.Blob` 35d and `Repository.read_obj()` 41b.

4.1.3 Reading a tree

⟨`Objects.read()` *match str cases* 42h⟩+≡ (41d) ◁42c 43g▷
```
| "tree"   -> Tree   (Tree.read ch2)
```
Uses `Objects.t.Tree` 35d and `Tree.read()` 183b.

⟨*signature* `Tree.read` 42i⟩≡ (183a)
```
(* assumes have already read the 'tree <size>\000' header from unzipped input *)
val read: IO.input -> t
```

⟨*function* `Tree.read` 42j⟩≡ (183b)
```
let read ch =
  let rec aux acc =
    try
      (* todo: how diffentiate no more input from wrong input ?
       * pass size ch and use IO.pos_in ?
       *)
      let e = read_entry ch in
      aux (e::acc)
    with IO.No_more_input ->
      List.rev acc
  in
  aux []
```
Uses `Tree.read_entry()` 43a.

⟨*function* `Tree.read_entry` 43a⟩≡ (183b)
```
(* todo: should transform some No_more_input exn in something bad,
 * on first one it's ok, but after it means incomplete entry.
 *)
let read_entry ch =
  let perm = IO_.read_string_and_stop_char ch ' ' in
  (* todo: handle escape char in filenames? encode/decode *)
  let name = IO_.read_string_and_stop_char ch '\000' in
  let hash = Sha1.read ch in
  { perm = perm_of_string perm; name = name; id = hash }
```
Uses `IO_.read_string_and_stop_char()` 146e, `Sha1.read()` 43c, `Tree.entry.id` 36b, `Tree.entry.name` 36b, `Tree.entry.perm` 36b, and `Tree.perm_of_string()` 183b.

⟨*signature* `Sha1.read` 43b⟩≡ (182a)
```
val read: IO.input -> t
```

⟨*function* `Sha1.read` 43c⟩≡ (182b)
```
let read ch =
  let s = IO.really_nread_string ch 20 in
  assert (is_sha s);
  s
```
Uses `IO.really_nread()` and `Sha1.is_sha()` 32d.

⟨*function* `Tree.perm_of_string` 43d⟩≡ (183b)
```
let perm_of_string = function
  | "44"
  | "100644" -> Normal
  | "100755" -> Exec
  | "120000" -> Link
  | "40000"  -> Dir
  ⟨Tree.perm_of_string() match str cases 132f⟩
  | x        -> failwith (spf "Tree.perm_of_string: %s is not a valid perm." x)
```
Uses `Tree.perm.Commit` 132a, `Tree.perm.Dir` 36d, `Tree.perm.Exec` 36d, `Tree.perm.Link` 36d, and `Tree.perm.Normal` 36d.

⟨*signature* `Repository.read_tree` 43e⟩≡ (179)
```
val read_tree: t -> Sha1.t -> Tree.t
```

⟨*function* `Repository.read_tree` 43f⟩≡ (180)
```
let read_tree r h =
  match read_obj r h with
  | Objects.Tree x -> x
  | _ -> failwith "read_tree: was expecting a tree"
```
Uses `Objects.t.Tree` 35d and `Repository.read_obj()` 41b.

4.1.4 Reading a commit

⟨`Objects.read()` *match str cases* 43g⟩+≡ (41d) ◁42h 134c▷
```
| "commit" -> Commit (Commit.read ch2)
```
Uses `Commit.read()` 44a and `Objects.t.Commit` 35d.

⟨*signature* `Commit.read` 43h⟩≡ (159a)
```
(* assumes have already read the 'commit <size>\000' hdr from unzipped input *)
val read: IO.input -> t
```

⟨*function* `Commit.read` 44a⟩≡ (159b)

```
let read ch =
  let tree =
    IO_.read_key_space_value_newline ch "tree" Hexsha.read in
  (* todo: read "parent" or "author", because first commit has no parent *)
  let parents, author =
    let rec loop parents =
      let str = IO_.read_string_and_stop_char ch ' ' in
      match str with
      | "parent" ->
        let v = Hexsha.read ch in
        let c = IO.read ch in
        if c <> '\n'
        then failwith "Commit.read: missing newline after parent";
        loop (v::parents)
      | "author" ->
        let v = User.read ch in
        let c = IO.read ch in
        if c <> '\n'
        then failwith "Commit.read: missing newline after author";
        List.rev parents, v
      | _ -> failwith (spf "Commit.read: was expecting parent or author not %s"
                           str)
    in
    loop []
  in
  let committer =
    IO_.read_key_space_value_newline ch "committer" User.read in
  let c = IO.read ch in
  if c <> '\n'
  then failwith "Commit.read: missing newline before message";
  let msg = IO.read_all ch in
  { tree = Hexsha.to_sha tree;
    parents = parents |> List.map Hexsha.to_sha;
    author = author; committer = committer;
    message = msg;
  }
```

Uses `Commit.t.author` 36e, `Commit.t.committer` 37c, `Commit.t.message` 36e, `Commit.t.parents` 36e, `Commit.t.tree` 36e, `Common.spf()`, `Hexsha.read()`, `Hexsha.to_sha()`, `IO.read()`, `IO.read_all()`, `IO_.read_key_space_value_newline()` 147, `IO_.read_string_and_stop_char()` 146e, and `User.read()` 44e.

⟨*signature* `Hexsha.read` 44b⟩≡ (172a)

```
val read: IO.input -> t
```

⟨*function* `Hexsha.read` 44c⟩≡ (172b)

```
let read ch =
  let s = IO.really_nread_string ch 40 in
  assert (is_hexsha s);
  s
```

Uses `Hexsha.is_hexsha()` and `IO.really_nread()`.

⟨*signature* `User.read` 44d⟩≡ (187a)

```
val read: IO.input -> t
```

⟨*function* `User.read` 44e⟩≡ (187b)

```
let read ch =
  let name = IO_.read_string_and_stop_char ch '<' in
  let email = IO_.read_string_and_stop_char ch '>' in
  let c = IO.read ch in
  if c <> ' ' then failwith "User.read: wrong format, missing space";
```

```
    let seconds = IO_.read_string_and_stop_char ch ' ' in
    let sign = IO.read ch in
    let hours = IO.nread_string ch 2 in
    let mins = IO.nread_string ch 2 in
    (* stricter: *)
    if int_of_string mins <> 0
    then failwith "User.read: timezeone with minutes not supported";

    { name = String.sub name 0 (String.length name - 1);
      email = email;
      date = (Int64.of_string seconds, (sign_of_char sign) (int_of_string hours));
    }
```

Uses IO.nread_string(), IO.read(), IO_.read_string_and_stop_char() 146e, User.sign_of_char() 45a, User.t.date 37a, User.t.email 37a, and User.t.name 37a.

⟨*function* User.sign_of_char 45a⟩≡ (187b)

```
  let sign_of_char = function
    | '+' -> (fun x -> x)
    | '-' -> (fun x -> - x)
    | c -> failwith (spf "User.sign_of_string: not a sign, got %c" c)
```

Uses Common.spf().

⟨*signature* Repository.read_commit 45b⟩≡ (179)

```
  val read_commit: t -> Sha1.t -> Commit.t
```

⟨*function* Repository.read_commit 45c⟩≡ (180)

```
  let read_commit r h =
    match read_obj r h with
    | Objects.Commit x -> x
    | _ -> failwith "read_commit: was expecting a commit"
```

Uses Objects.t.Commit 35d and Repository.read_obj() 41b.

4.2 References

References use a simpler format, and are stored in plain text.

⟨*signature* Repository.read_ref 45d⟩≡ (179)

```
  val read_ref: t -> Refs.t -> Refs.ref_content
```

⟨*function* Repository.read_ref 45e⟩≡ (180)

```
  let read_ref r aref =
    (* less: packed refs *)
    let file = ref_to_filename r aref in
    file |> UChan.with_open_in (fun (ch : Chan.i) ->
      ch.ic |> IO.input_channel |> Refs.read
    )
```

Uses Common.with_file_in(), IO.input_channel(), Refs.read(), and Repository.ref_to_filename() 38e.

⟨*signature* Refs.read 45f⟩≡ (178a)

```
  val read: IO.input -> ref_content
```

⟨*function* Refs.read 45g⟩≡ (178b)

```
  let read ch =
    let str = IO.read_all ch in
    (* less: check finish by newline? *)
    match str with
    | _ when str =~ "^ref: \\(.*\\)$" -> OtherRef (Regexp_.matched1 str)
    | _ -> Hash (str |> IO.input_string |> Hexsha.read |> Hexsha.to_sha)
```

Uses Common.Regexp_.matched1(), Hexsha.read(), Hexsha.to_sha(), IO.input_string(), IO.read_all(), Refs.ref_content.Hash 38b, and Refs.ref_content.OtherRef 38b.

Most of the time, you want to follow a reference until you get an SHA1 hash.

⟨*signature* Repository.follow_ref 46a⟩≡ (179)
```
val follow_ref: t -> Refs.t -> Refs.t list * Commit.hash option
```

⟨*function* Repository.follow_ref 46b⟩≡ (180)
```
let rec follow_ref r aref =
  (* less: check if depth > 5? *)
  try (
  let content = read_ref r aref in
  match content with
  | Refs.Hash sha -> [aref], Some sha
  | Refs.OtherRef refname ->
    let (xs, shaopt) = follow_ref r (Refs.Ref refname) in
    aref::xs, shaopt
  )
  (* inexistent ref file, can happen at the beginning when have .git/HEAD
   * pointing to an inexistent .git/refs/heads/master
   *)
  with Sys_error _ (* no such file or directory *) -> [aref], None
```
Uses Refs.ref_content.Hash 38b, Refs.ref_content.OtherRef 38b, Refs.t.Ref 37g, Repository.follow_ref() 46b, and Repository.read_ref() 45e.

⟨*signature* Repository.follow_ref_some 46c⟩≡ (179)
```
val follow_ref_some: t -> Refs.t -> Commit.hash
```

⟨*function* Repository.follow_ref_some 46d⟩≡ (180)
```
let follow_ref_some r aref =
  match follow_ref r aref |> snd with
  | Some sha -> sha
  | None -> failwith (spf "could not follow %s" (Refs.string_of_ref aref))
```
Uses Common.spf(), Refs.string_of_ref(), and Repository.follow_ref() 46b.

⟨*signature* Repository.read_objectish 46e⟩≡ (179)
```
val read_objectish: t -> objectish -> Sha1.t * Objects.t
```

⟨*function* Repository.read_objectish 46f⟩≡ (180)
```
let rec read_objectish r objectish =
  match objectish with
  | ObjByRef aref ->
    (match follow_ref r aref |> snd with
    | None -> failwith (spf "could not resolve %s" (Refs.string_of_ref aref))
    | Some sha ->
      sha, read_obj r sha
    )
  | ObjByHex hexsha ->
    let sha = Hexsha.to_sha hexsha in
    sha, read_obj r sha
  | ObjByBranch str ->
    read_objectish r (ObjByRef (Refs.Ref ("refs/heads/" ^ str)))
```
Uses Common.spf(), Hexsha.to_sha(), Refs.string_of_ref(), Refs.t.Ref 37g, Repository.follow_ref() 46b, Repository.objectish.ObjByBranch 38f, Repository.objectish.ObjByHex 38f, Repository.objectish.ObjByRef 38f, Repository.read_obj() 41b, and Repository.read_objectish() 46f.

4.3 Index

⟨*signature* Repository.read_index 46g⟩≡ (179)
```
val read_index: t -> Index.t
```

⟨*function* `Repository.read_index` 47a⟩≡ (180)
```
let read_index r =
  r.index
```
Uses `Repository.t.index`.

⟨*signature* `Index.read` 47b⟩≡ (173a)
```
val read: IO.input -> t
```

⟨*function* `Index.read` 47c⟩≡ (173b)
```
let read ch =
  let header = IO.really_nread_string ch 4 in
  if header <> "DIRC"
  then failwith "Index.read: expecting DIRC header";
  let version = IO.BigEndian.read_i32 ch in
  if version <> 2
  then failwith "Index.read: expecting version 2";
  let entries = read_entries ch in
  (* todo: read_extensions but need know when reach last 20 bytes *)
  (* todo: check hash correctly stored in last 20 bytes *)
  entries
```
Uses `IO.nwrite()`, `Index.write_entry()` 47e, `Sha1.sha1()`, and `Sha1.write()` 51f.

⟨*function* `Index.read_entries` 47d⟩≡ (173b)
```
let read_entries ch =
  let n = IO.BigEndian.read_i32 ch in
  let rec loop acc n =
    if n = 0
    then List.rev acc
    else
      let entry = read_entry ch in
      loop (entry :: acc) (n - 1) in
  loop [] n
```
Uses `IO.output_bytes()` and `IO_.with_close_out()` 146c.

⟨*function* `Index.read_entry` 47e⟩≡ (173b)
```
let read_entry (ch : IO.input) : entry =
  let stats = read_stat_info ch in
  let id = Sha1.read ch in
  let stage, len =
    let i = IO.BigEndian.read_ui16 ch in
    (i land 0x3000) lsr 12,
    (i land 0x0FFF)
  in
  if (stage <> 0)
  then failwith (spf "stage is not 0: %d" stage);
  let path = IO.really_nread_string ch len in
  let c = IO.read ch in
  if c <> '\000'
  then failwith "Index.read_entry: expecting null char after name";
  let len = 63 + String.length path in
  let pad =
    match len mod 8 with
    | 0 -> 0
    | n -> 8-n
  in
  let _zeros = IO.really_nread ch pad in
  (* less: assert zeros *)
  { stats = stats; id = id; path = Fpath.v path }
```
Uses `IO.BigEndian.write_ui16()`, `IO.nwrite()`, `IO.write()`, `Index.entry.id` 39c, `Index.entry.path` 39c, `Index.entry.stats` 39c, `Index.write_stat_info()`, and `Sha1.write()` 51f.

⟨*function* `Index.read_stat_info` 48a⟩≡ (173b)

```
let read_stat_info ch =
  let ctime = read_time ch in
  let mtime = read_time ch in
  (* less: unsigned again *)
  let dev = (*IO.BigEndian.*)read_real_i32 ch in
  let inode = (*IO.BigEndian.*)read_real_i32 ch in
  let mode = read_mode ch in
  let uid = (*IO.BigEndian.*)read_real_i32 ch in
  let gid = (*IO.BigEndian.*)read_real_i32 ch in
  let size = (*IO.BigEndian.*)read_real_i32 ch in
  { mtime = mtime; ctime = ctime; dev = dev; inode = inode; mode = mode; uid = uid; gid = gid; size = size }
```

Uses `IO.BigEndian.read_ui16()`, `IO.BigEndian.write_real_i32()`, `Index.read_stat_info()` 48c, `Index.stat_info.size` 39d, and `Sha1.read()` 43c.

⟨*function* `Index.read_time` 48b⟩≡ (173b)

```
let read_time ch =
  (* less: unsigned actually *)
  let lsb32 = (*IO.BigEndian.*)read_real_i32 ch in
  let nsec = (*IO.BigEndian.*)read_real_i32 ch in
  { lsb32 = lsb32; nsec = nsec }
```

Uses `Common.spf()`, `Index.mode.Exec` 39e, and `Index.mode.Normal` 39e.

⟨*function* `Index.read_mode` 48c⟩≡ (173b)

```
let read_mode ch =
  let _zero = IO.BigEndian.read_ui16 ch in
  let n = IO.BigEndian.read_ui16 ch in
  match n lsr 12 with
  | 0b1010 -> Link
  ⟨Index.read_mode() match n lsr 12 cases 133a⟩
  | 0b1000 ->
    (match n land 0x1FF with
    | 0o755 -> Exec
    | 0o644 -> Normal
    | d     -> failwith (spf "Index.mode: invalid permission (%d)" d)
    )
  | m -> failwith (spf "Index.mode: invalid (%d)" m)
```

Uses `Index.mode.Exec` 39e, `Index.mode.Gitlink` 132b, `Index.mode.Link` 39e, `Index.mode.Normal` 39e, and `Index.read_time()` 173b.

Chapter 5

Writing to a Repository

In this chapter, you will see the code of `Repository.add_obj()`[49b], `Repository.write_ref()`[52g], and `Repository.wr`
Those functions just do the reverse operations of functions you have seen in Chapter 4.

5.1 Objects

⟨*signature* `Repository.add_obj` 49a⟩≡ (179)
```
val add_obj: t -> Objects.t -> Sha1.t
```

⟨*function* `Repository.add_obj` 49b⟩≡ (180)
```
let add_obj r obj =
  let bytes =
    IO.output_bytes () |> IO_.with_close_out (Objects.write obj) in
  let sha = Sha1.sha1 (Bytes.to_string bytes) in
  let hexsha = Hexsha.of_sha sha in
  let file = hexsha_to_filename r hexsha in
  ⟨Repository.add_obj() create directory if it does not exist 49c⟩
  if (Sys.file_exists !!file)
  then sha (* deduplication! nothing to write, can share objects *)
  else begin
    file |> with_file_out_with_lock (fun ch ->
      let ic = IO.input_bytes bytes in
      let oc = IO.output_channel ch in
      Compression.compress ic oc;
      IO.close_out oc;
    );
    sha
  end
```
Uses `Compression.compress()` 50c, `Hexsha.of_sha()`, `IO.close_out()`, `IO.input_bytes()`, `IO.output_bytes()`, `IO.output_channel()`, `IO_.with_close_out()` 146c, `Objects.write()` 50a, `Repository.hexsha_to_filename()` 35b, `Repository.with_file_out_with_lock()` 50d, and `Sha1.sha1()`.

⟨`Repository.add_obj()` *create directory if it does not exist* 49c⟩≡ (49b)
```
let dir = Filename.dirname !!file in
if not (Sys.file_exists dir)
then Unix.mkdir dir dirperm;
```
Uses `Repository.dirperm` 60a.

⟨*signature* `Objects.write` 49d⟩≡ (177b)
```
(* will not compress, will return unserialized content for sha1 computation *)
val write: t -> bytes IO.output -> unit
```

⟨*function* `Objects.write` 50a⟩≡ (177c)
```
let write obj ch =
  let body =
    IO.output_bytes () |> IO_.with_close_out (fun ch ->
      match obj with
      | Blob x   -> Blob.write x ch
      | Commit x -> Commit.write x ch
      | Tree x   -> Tree.write x ch
      ⟨Objects.write() match obj cases 134d⟩
    )
  in
  let header =
    spf "%s %d\000"
      (match obj with
      | Blob _   -> "blob"
      | Commit _ -> "commit"
      | Tree  _  ->  "tree"
      ⟨Objects.write() return header, match obj cases 134e⟩
      )
      (Bytes.length body)
  in
  IO.nwrite_string ch header;
  IO.nwrite ch body
```
Uses `Blob.write()` 51a, `Commit.write()` 51i, `Common.spf()`, `IO.nwrite()`, `IO.output_bytes()`, `IO_.with_close_out()` 146c, `Objects.t.Blob` 35d, `Objects.t.Commit` 35d, `Objects.t.Tree` 35d, and `Tree.write()` 183b.

5.1.1 Compression

⟨*signature* `Compression.compress` 50b⟩≡ (160a)
```
val compress:
  IO.input -> 'a IO.output -> unit
```

⟨*function* `Compression.compress` 50c⟩≡ (160b)
```
let compress ic oc =
  Zlib.compress
    (fun buf ->
      try IO.input ic buf 0 (Bytes.length buf)
      with IO.No_more_input -> 0
    )
    (fun buf len ->
      IO.output oc buf 0 len |> ignore)
```
Uses `IO.input()` and `Zlib.compress()` 126d.

5.1.2 Locking

⟨*function* `Repository.with_file_out_with_lock` 50d⟩≡ (180)
```
(* todo: see code of _Gitfile.__init__ O_EXCL ... *)
let with_file_out_with_lock f (file : Fpath.t) =
  (* todo: create .lock file and then rename *)
  UChan.with_open_out (fun (chan : Chan.o) -> f chan.oc) file
```
Uses `Common.with_file_out()`.

5.1.3 Writing a blob

⟨*signature* `Blob.write` 50e⟩≡ (148c)
```
(* does not write the header, does not compress *)
val write: t -> bytes IO.output -> unit
```

⟨*function* `Blob.write` 51a⟩≡ (149a)
```
let write blob ch =
  IO.nwrite_string ch blob
```

5.1.4 Writing a tree

⟨*signature* `Tree.write` 51b⟩≡ (183a)
```
(* does not write the header, does not compress *)
val write: t -> bytes IO.output -> unit
```

⟨*function* `Tree.write` 51c⟩≡ (183b)
```
let write t ch =
  t |> List.iter (write_entry ch)
```

⟨*function* `Tree.write_entry` 51d⟩≡ (183b)
```
let write_entry ch e =
  IO.nwrite_string ch (string_of_perm e.perm);
  IO.write ch ' ';
  (* todo: handle escape char in filenames? encode/decode *)
  IO.nwrite_string ch e.name;
  IO.write ch '\000';
  Sha1.write ch e.id
```
Uses `IO.nwrite()`, `IO.write()`, `Sha1.write()` 51f, `Tree.entry.id` 36b, and `Tree.entry.name` 36b.

⟨*signature* `Sha1.write` 51e⟩≡ (182a)
```
val write: 'a IO.output -> t -> unit
```

⟨*function* `Sha1.write` 51f⟩≡ (182b)
```
let write ch x =
  IO.nwrite_string ch x
```

⟨*function* `Tree.string_of_perm` 51g⟩≡ (183b)
```
let string_of_perm = function
  | Normal -> "100644"
  | Exec   -> "100755"
  | Link   -> "120000"
  | Dir    -> "40000"
  ⟨Tree.string_of_perm() match perm cases 132g⟩
```
Uses `Tree.perm.Commit` 132a, `Tree.perm.Dir` 36d, `Tree.perm.Exec` 36d, and `Tree.perm.Link` 36d.

5.1.5 Writing a commit

⟨*signature* `Commit.write` 51h⟩≡ (159a)
```
(* does not write the header, does not compress *)
val write: t -> 'a IO.output -> unit
```

⟨*function* `Commit.write` 51i⟩≡ (159b)
```
let write commit ch =
  IO.nwrite_string ch "tree ";
  Hexsha.write ch (Hexsha.of_sha commit.tree);
  IO.write ch '\n';
  commit.parents |> List.iter (fun parent ->
    IO.nwrite_string ch "parent ";
    Hexsha.write ch (Hexsha.of_sha parent);
    IO.write ch '\n';
  );
  IO.nwrite_string ch "author ";
  User.write ch commit.author;
```

```
    IO.write ch '\n';
    IO.nwrite_string ch "committer ";
    User.write ch commit.committer;
    IO.write ch '\n';

    IO.write ch '\n';
    IO.nwrite_string ch commit.message
```

Uses Commit.t.author 36e, Commit.t.committer 37c, Commit.t.message 36e, Commit.t.parents 36e, Commit.t.tree 36e, Hexsha.of_sha(), Hexsha.write(), IO.nwrite(), IO.write(), and User.write() 52d.

⟨*signature* Hexsha.write 52a⟩≡ (172a)
```
  val write: 'a IO.output -> t -> unit
```

⟨*function* Hexsha.write 52b⟩≡ (172b)
```
  let write ch x =
    IO.nwrite_string ch x
```

Uses IO.nwrite().

⟨*signature* User.write 52c⟩≡ (187a)
```
  val write: 'a IO.output -> t -> unit
```

⟨*function* User.write 52d⟩≡ (187b)
```
  let write ch user =
    IO.nwrite_string ch (spf "%s <%s> " user.name user.email);
    write_date ch user.date
```

Uses Common.spf(), IO.nwrite(), User.t.date 37a, User.t.email 37a, User.t.name 37a, and User.write_date() 52e.

⟨*function* User.write_date 52e⟩≡ (187b)
```
  let write_date ch (date, tz) =
    IO.nwrite_string ch (Int64.to_string date);
    IO.write ch ' ';
    IO.nwrite_string ch (spf "%c%02d%02d" (char_of_sign tz) (abs tz) 0)
```

Uses Common.spf(), IO.nwrite(), IO.write(), and User.char_of_sign() 84c.

5.2 References

⟨*signature* Repository.write_ref 52f⟩≡ (179)
```
  val write_ref: t -> Refs.t -> Refs.ref_content -> unit
```

⟨*function* Repository.write_ref 52g⟩≡ (180)
```
  (* low-level *)
  let write_ref r aref content =
    let file = ref_to_filename r aref in
    file |> with_file_out_with_lock (fun ch ->
      ch |> IO.output_channel |> IO_.with_close_out (Refs.write content))
```

Uses IO.output_channel(), IO_.with_close_out() 146c, Refs.write(), Repository.ref_to_filename() 38e, and Repository.with_file_out_with_lock() 50d.

⟨*signature* Refs.write 52h⟩≡ (178a)
```
  val write: ref_content -> unit IO.output -> unit
```

⟨*function* Refs.write 52i⟩≡ (178b)
```
  let write content ch =
    match content with
    | Hash h ->
      IO.nwrite_string ch (Hexsha.of_sha h ^ "\n")
    | OtherRef name ->
      IO.nwrite_string ch ("ref: " ^ name ^ "\n")
```

Uses Hexsha.of_sha(), IO.nwrite_string(), Refs.ref_content.Hash 38b, and Refs.ref_content.OtherRef 38b.

5.3 Index

⟨*signature* Repository.write_index 53a⟩≡ (179)
```
val write_index: t -> unit
```

⟨*function* Repository.write_index 53b⟩≡ (180)
```
let write_index r =
  let path = index_to_filename r in
  path |> with_file_out_with_lock (fun ch ->
    ch |> IO.output_channel |> IO_.with_close_out (Index.write r.index)
  )
```
Uses IO.output_channel(), IO_.with_close_out() 146c, Index.write() 47d, Repository.index_to_filename() 39a, Repository.t.index, and Repository.with_file_out_with_lock() 50d.

⟨*signature* Index.write 53c⟩≡ (173a)
```
(* will write the header, and sha checksum at the end *)
val write: t -> unit IO.output -> unit
```

⟨*function* Index.write 53d⟩≡ (173b)
```
let write idx ch =
  let n = List.length idx in
  let body =
    IO.output_bytes () |> IO_.with_close_out (fun ch ->
      IO.nwrite_string ch "DIRC";
      IO.BigEndian.write_i32 ch 2;
      IO.BigEndian.write_i32 ch n;
      idx |> List.iter (write_entry ch)
    )
  in
  let sha = Sha1.sha1 (Bytes.to_string body) in
  IO.nwrite ch body;
  Sha1.write ch sha
```

⟨*function* Index.write_entry 53e⟩≡ (173b)
```
let write_entry ch (e : entry) =
  write_stat_info ch e.stats;
  Sha1.write ch e.id;
  let flags = (0 lsl 12 + String.length !!(e.path)) land 0x3FFF in
  IO.BigEndian.write_ui16 ch flags;
  IO.nwrite_string ch !!(e.path);
  let len = 63 + String.length !!(e.path) in
  let pad =
    match len mod 8 with
    | 0 -> 0
    | n -> 8-n
  in
  IO.nwrite ch (Bytes.make pad '\000');
  IO.write ch '\000'
```
Uses IO.really_nread_string() and Index.read_entry() 48a.

⟨*function* Index.write_stat_info 53f⟩≡ (173b)
```
let write_stat_info ch stats =
  write_time ch stats.ctime;
  write_time ch stats.mtime;
  (*IO.BigEndian.*)write_real_i32 ch stats.dev;
  (*IO.BigEndian.*)write_real_i32 ch stats.inode;
  write_mode ch stats.mode;
  (*IO.BigEndian.*)write_real_i32 ch stats.uid;
  (*IO.BigEndian.*)write_real_i32 ch stats.gid;
  (*IO.BigEndian.*)write_real_i32 ch stats.size;
  ()
```
Uses Common.spf(), IO.read(), and IO.really_nread_string().

⟨*function* `Index.write_mode` 54a⟩≡ (173b)

```
let write_mode ch mode =
  IO.BigEndian.write_ui16 ch 0;
  let n =
    match mode with
    | Exec    -> 0b1000__000__111_101_101
    | Normal  -> 0b1000__000__110_100_100
    | Link    -> 0b1010__000__000_000_000
    ⟨Index.write_mode() match mode cases 133b⟩
  in
  IO.BigEndian.write_ui16 ch n
```

Uses `IO.BigEndian.read_real_i32()`, `IO.BigEndian.write_real_i32()`, `Index.stat_info.ctime` 39d, `Index.stat_info.dev` 39d, `Index.stat_info.gid` 39d, `Index.stat_info.inode` 39d, `Index.stat_info.mode` 39d, `Index.stat_info.mtime` 39d, `Index.stat_info.size` 39d, and `Index.stat_info.uid` 39d.

⟨*function* `Index.write_time` 54b⟩≡ (173b)

```
let write_time ch time =
  (*IO.BigEndian.*)write_real_i32 ch time.lsb32;
  (*IO.BigEndian.*)write_real_i32 ch time.nsec
```

Chapter 6

Main Functions

I now switch from the bottom-up approach started in Chapter 3 to a top-down approach, starting in this chapter with the main entry point of `ocamlgit`: `Main.main()`[56b]. The next chapters will then describe the code of the main Git commands.

6.1 Main commands

`ocamlgit` is a single program, but it consists really of a set of independent mini-programs: the different Git commands. Each of those mini-programs, like `ocamlgit` itself, have a name, some usage documentation, and a set of command-line options, hence the following type:

⟨*type* `Cmd_.t` 55a⟩≡ (153a)
```
type t = {
  name: string;
  usage: string;
  options: (Arg.key * Arg.spec * Arg.doc) list;

  (* the command! *)
  f: string list -> unit;
  (* less: man: when do git -help get short help, and with --help man page *)
}
```

The last field of `Cmd_.t`[55a] is the callback function to call to execute the Git command. It takes as a parameter the list of command-line arguments that were not flags.

The constant below contains the list of all the main Git commands. Each module defines its own `cmd` constant:

⟨*constant* `Cmds.main_commands` 55b⟩≡ (158)
```
let main_commands = [
  (* creating *)
  Cmd_init.cmd;
  Cmd_add.cmd;
  Cmd_rm.cmd;
  Cmd_commit.cmd;

  (* branching *)
  Cmd_branch.cmd;
  Cmd_checkout.cmd;
  Cmd_reset.cmd;

  (* inspecting *)
  Cmd_show.cmd;
  Cmd_diff.cmd;
  Cmd_log.cmd;
```

```
    Cmd_status.cmd;

    (* networking *)
    Cmd_pull.cmd;
    Cmd_push.cmd;
    Cmd_clone.cmd;
  ]
```

Uses Cmd_add.cmd 62a, Cmd_branch.cmd 153c, Cmd_checkout.cmd 76c, Cmd_clone.cmd 106a, Cmd_commit.cmd 66a, Cmd_diff.cmd 85d, Cmd_init.cmd 59a, Cmd_log.cmd, Cmd_pull.cmd 99d, Cmd_push.cmd, Cmd_reset.cmd 79a, Cmd_rm.cmd 64b, Cmd_show.cmd 82a, and Cmd_status.cmd 156d.

I will present gradually in the rest of the document the code of all those commands.

6.2 Main.main()

I can finally present the entry point of ocamlgit, which will dispatch one of the Git commands.

⟨*toplevel* Main._1 56a⟩≡ (176)

```
let _ =
  main ()
```

⟨*function* Main.main 56b⟩≡ (176)

```
let main () =
  ⟨Main.main() GC settings 143⟩
  ⟨Main.main() sanity check arguments 56d⟩
  else begin
    let cmd =
      try
        commands |> List.find (fun cmd -> cmd.Cmd_.name = Sys.argv.(1))
      with Not_found ->
        ⟨Main.main() print usage and exit 56e⟩
    in
    ⟨Main.main() execute cmd.f 57b⟩
  end
```

Uses Cmd.t.name 55a, Common.pr(), Common.spf(), version_control/Main.commands 176, and version_control/Main.usage().

The final list of commands below contains the main commands (see Section 6.1 above), some extra commands to either help debug ocamlgit (see Appendix A) or to provide advanced features (see Chapter 18), and the command to get help (see Section 6.3).

⟨*constant* Main.commands 56c⟩≡ (176)

```
let commands = List.flatten [
  Cmds.main_commands;
  Cmds.extra_commands;
  [Cmd_help.cmd];
]
```

Uses Cmd_help.cmd 58.

ocamlgit requires at least one argument: a Git command. Thus, argv must contain at least two elements (remember that argv.(0) contains the name of the program, here "ocamlgit"):

⟨Main.main() *sanity check arguments* 56d⟩≡ (56b)

```
if Array.length Sys.argv < 2
then begin
  ⟨Main.main() print usage and exit 56e⟩
end
```

Uses Common.pr() and version_control/Main.usage().

⟨Main.main() *print usage and exit* 56e⟩≡ (56)

```
UConsole.print (usage ());
exit 1
```

⟨*function* `Main.usage` 57a⟩≡ (176)
```
let usage () =
  spf "usage: ocamlgit <%s> [options]"
    (String.concat "|" (commands |> List.map (fun cmd -> cmd.Cmd_.name)))
```

The code to sanity check arguments above is necessary but often obvious. This is why I will usually not comment such code in the rest of this document.

Executing the command callback requires first to build a new `argv` (without the leading `"ocamlgit"`), to have a new usage message specific to the command, and finally to process the new `argv` to separate flags and options from the remaining arguments of the command:

⟨`Main.main()` *execute* `cmd.f` 57b⟩≡ (56b)
```
let argv = Array.sub Sys.argv 1 (Array.length Sys.argv -1) in
let usage_msg_cmd = spf "usage: %s %s%s"
  (Filename.basename Sys.argv.(0))
  cmd.Cmd_.name
  cmd.Cmd_.usage
in
let remaining_args = ref [] in
⟨Main.main() parse argv for cmd options and remaining_args 57d⟩
(* finally! *)
try
  cmd.Cmd_.f (List.rev !remaining_args)
with
  | Cmd_.ShowUsage ->
    Arg.usage (Arg.align cmd.Cmd_.options) usage_msg_cmd;
    exit 1
```
Uses `Cmd.t.name` 55a, `Cmd.t.options` 55a, and `Cmd.t.usage` 55a.

The exception below will be used from command callbacks to indicate that something was wrong with the command-line arguments:

⟨*exception* `Cmd_.ShowUsage` 57c⟩≡ (153a)
```
(* Cmd_.f can raise ShowUsage. It will be catched by Main.main *)
exception ShowUsage
```

Processing the new `argv` is similar to what most OCaml programs do and it involves the `Arg` module:

⟨`Main.main()` *parse* `argv` *for cmd options and* `remaining_args` 57d⟩≡ (57b)
```
(try
 (* todo: look if --help and factorize treatment of usage for subcmds *)
   Arg.parse_argv argv (Arg.align cmd.Cmd_.options)
     (fun arg -> Stack_.push arg remaining_args) usage_msg_cmd;
 with Arg.Bad str | Arg.Help str->
   prerr_string str;
   exit 1
);
```
Uses `Common.push()`.

6.3 Getting help: `git help`

Here is the code of the first Git command, the command to get help with `git help`:

⟨*constant* `Cmd_help.list_extra` 57e⟩≡ (154e)
```
let list_extra = ref false
```

⟨*constant* `Cmd_help.cmd` 58⟩≡ (154e)

```
let rec cmd = { Cmd_.
  name = "help";
  usage = "";
  options = ["-a", Arg.Set list_extra, " see all commands"];
  f = (fun _args ->
    let xs =
      if !list_extra
      then Cmds.main_commands @ Cmds.extra_commands @ [cmd]
      else Cmds.main_commands
    in
    UConsole.print ("Available commands: ");
    xs |> List.iter (fun cmd ->
      UConsole.print (spf "  %s" cmd.Cmd_.name);
    );
  );
}
```

Uses `Cmd.t.f` 55a, `Cmd.t.name` 55a, `Cmd.t.options` 55a, `Cmd.t.usage` 55a, `Cmd_help.cmd` 58, `Cmd_help.list_extra` 57e, `Cmds.extra_commands` 140a, `Cmds.main_commands` 55b, `Common.pr()`, and `Common.spf()`.

Note that this command references itself, hence the `rec` keyword above. Moreover, this command also references `Cmds.main_commands`[55b]; this is why `Cmd_help.cmd` could not be part of the list of main commands earlier in Section 6.1 (OCaml does not allow mutually recursive modules).

Here is the output of `git help`:

```
1 $ git help
Available commands:
  init
  add
  rm
  commit
  branch
  checkout
  reset
  show
  diff
  log
  status
  pull
  push
  clone
```

Chapter 7

Creating a Repository

In this chapter, we will explore the different ways to create a Git repository.

7.1 Initializing a new repository: `git init`

The simplest way to create a new repository is to run the command `git init` from a directory, as shown in Section 1.4. Here is the code describing this command:

⟨*constant* `Cmd_init.cmd` 59a⟩≡ (155a)
```
let cmd = { Cmd_.
  name = "init";
  usage = " [directory]";
  options = [
   ⟨Cmd_init.cmd command-line options ??⟩
  ];
  f = (fun args ->
    match args with
    | []    -> Repository.init (Fpath.v ".")
    | [dir] -> Repository.init (Fpath.v dir)
    | _ -> raise Cmd_.ShowUsage
  );
}
```
Uses `Cmd.t.f` 55a, `Cmd.t.name` 55a, `Cmd.t.usage` 55a, and `Repository.init()` 59c.

As I explained in Section 2.5.1, initializing a fresh repository consists mainly in creating a directory structure under `.git/`, without any objects in it.

⟨*signature* `Repository.init` 59b⟩≡ (179)
```
val init: Fpath.t -> unit
```

⟨*function* `Repository.init` 59c⟩≡ (180)
```
let init (root : Fpath.t) =
  if not (Sys.file_exists !!root)
  then Unix.mkdir !!root dirperm;

  (* less: bare argument? so no .git/ prefix? *)
  let dirs = [
    ".git";
    ".git/objects";
    ".git/refs";
    ".git/refs/heads";
    ".git/refs/tags";
    ".git/refs/remote";
    ".git/refs/remote/origin";
    ".git/hooks";
```

```
      ".git/info";
    ] in
    dirs |> List.iter (fun dir ->
      (* less: exn if already there? *)
      Unix.mkdir !!(root / dir) dirperm;
    );
    ⟨Repository.init() create .git/HEAD 60b⟩

    (* less: config file, description, hooks, etc *)
    Sys.chdir !!root;
    let absolute = Sys.getcwd () |> Fpath.v in
    UConsole.print (spf "Initialized empty Git repository in %s" !!(absolute / ".git/"))
```

Uses Common.pr(), Common.spf(), Repository./(), and Repository.dirperm 60a.

⟨*constant* Repository.dirperm 60a⟩≡ (180)

```
  (* rwxr-x--- *)
  let dirperm = 0o750
```

`git init` also creates the special reference `HEAD` stored in `.git/HEAD`, and initializes its content with a reference to the master branch (see the content of `Refs.default_head_content`[38d]).

⟨Repository.init() *create* .git/HEAD 60b⟩≡ (59c)

```
  let r = {
    worktree = root;
    dotgit = root / ".git";
    index = Index.empty;
  } in
  add_ref_if_new r Refs.Head Refs.default_head_content |> ignore;
```

Uses Index.empty 173b, Refs.default_head_content, Refs.t.Head 37g, Repository./(), Repository.add_ref_if_new() 71d, Repository.t.dotgit 35a, Repository.t.index, and Repository.t.worktree 35a.

I will explain the code of `Repository.add_ref_if_new()`[71d] later and why it has `_if_new` in its name.

7.2 Copying an existing repository: `cp -r`

Another way to create a repository is to copy an existing one with a simple command such as `cp -r`. This is also an easy way to create a new branch, even though it is not an optimal one (space and speed-wise). You will see a more efficient way to manage branches in Chapter 10.

7.3 Cloning an existing repository: `git clone`

Using `cp -r` requires to have access to the original repository through the filesystem, which usually limits yourself to local repositories on your disk (unless you use a distributed filesystem such as NFS). A more flexible way to copy an existing repository is to use the command `git clone`, which accepts a URL as an argument. Here is the command to clone the repository containing the code of `ocamlgit` itself:

```
$ git clone http://github.com/aryx/plan9-ocaml
Cloning into 'plan9-ocaml'...
$ cd plan9-ocaml/version-control/
$ ls
IO_.ml            cmd_checkout.ml  diff.mli          refs.ml
...
```

Cloning a repository does not just copy it; it also keeps a reference to the remote repository (in `refs/remotes/orig`
This reference will allow later to *push* and *pull* updates from the repository (see Chapter 14).

I will describe the code of `git clone` later in Section 14.5, after I describe the client/server architecture of Git and the code to pack objects together in Chapter 13.

7.4 Opening a repository

The Git commands in the following chapters all assume they are run from an existing repository. Here is the function to open a repository and get a `Repository.t`[35a] record:

⟨*signature* `Repository.open_` 61a⟩≡ (179)
```
val open_: Fpath.t -> t
```

⟨*function* `Repository.open_` 61b⟩≡ (180)
```
let open_ (root : Fpath.t) =
  let path = root / ".git" in
  if Sys.file_exists !!path &&
     (Unix.stat !!path).Unix.st_kind = Unix.S_DIR
  then
    { worktree = root;
      dotgit = path;
      ⟨Repository.open_() other fields settings 61c⟩
    }
  else failwith (spf "Not a git repository at %s" !!root)
```
Uses `Common.spf()`, `Repository./()`, `Repository.t.dotgit` 35a, and `Repository.t.worktree` 35a.

As I explained in Section 3.5, most Git commands will also need to read or write information in the index, so opening a repository also reads its index:

⟨`Repository.open_()` *other fields settings* 61c⟩≡ (61b) ?? ▷
```
index =
  (if Sys.file_exists !!(path / "index")
   then
    (path / "index") |> UChan.with_open_in (fun (ch : Chan.i) ->
      ch.ic |> IO.input_channel |> Index.read)
   else Index.empty
  );
```
Uses `Common.with_file_in()`, `IO.input_channel()`, `Index.empty` 173b, `Index.read()` 53e, `Repository./()`, and `Repository.t.index`.

I described before the code of `Index.read()`[53e] called above.

It is convenient for the user to be able to run a Git command from anywhere inside the repository, not just from its root. However, it is simpler, while implementing the code of the different Git commands, to assume that every paths are relative to the root of the repository. Avoiding variation in data usually simplifies things. Moreover, some data structures such as the index entries must contain relative paths. The function below allows to satisfy both requirements by converting any filenames given on the command-line to relative filenames, and to find the root of the repository:

⟨*signature* `Repository.find_dotgit_root_and_open` 61d⟩≡ (179)
```
val find_root_open_and_adjust_paths:
 Fpath.t list -> t * Fpath.t list
```

⟨*function* `Repository.find_dotgit_root_and_open` 61e⟩≡ (180)
```
let find_root_open_and_adjust_paths (paths : Fpath.t list) : t * Fpath.t list =
  (* todo: allow git from different location *)
  let r = open_ (Fpath.v ".") in
  (* todo: support also absolute paths and transform in relpaths *)
  let relpaths = paths |> List.map (fun path ->
    if Filename.is_relative !!path
    then
      (* todo: may have to adjust if root was not pwd *)
      path
    else failwith (spf "TODO: Not a relative path: %s" !!path)
    )
  in
  r, relpaths
```
Uses `Common.spf()` and `Repository.open_()` 61b.

Chapter 8

Staging a Diff

Once you created a repository, you can add, remove, modify, or rename files in this repository, as explained in the following sections. In Git, doing those modifications requires two steps: first you stage your modification, and then you commit the modification (which gives flexibility in the commit process, as explained in Section 2.3.3). This chapter is concerned only with the first part: the staging. The next chapter will cover the commit.

8.1 Adding files: `git add`

Here is the code of the command to add a file or set of files to the staging area:

⟨*constant* `Cmd_add.cmd` 62a⟩≡ (153b)

```
let cmd = { Cmd_.
  name = "add";
  usage = " <file>..."; (* less: pathspec? *)
  options = [
    ⟨Cmd_add.cmd command-line options ??⟩
  ];
  f = (fun args ->
    match args with
    | [] -> Logs.app (fun m -> m "Nothing specified, nothing added.")
    | xs ->
      let r, relpaths = Repository.find_root_open_and_adjust_paths (Fpath_.of_strings xs) in
      (* less: support directories *)
      add r relpaths
  );
}
```

Uses `Cmd.t.f` 55a, `Cmd.t.name` 55a, `Cmd.t.usage` 55a, `Cmd_add.add()` 62b, `Common.pr2()`, and `Repository.find_root_open_and_adjust` 61e.

Note that you must use `git add` both when adding a new file to the repository and when adding a new version of a file that was already in the repository. In some senses, Git does not care whether the file is new or not. In both cases, Git just adds the current version of a file in the staging area.

Adding files to the staging area translates internally in adding them to the index:

⟨*function* `Cmd_add.add` 62b⟩≡ (153b)

```
let add r relpaths =
  (* this will also add some blobs to the object store *)
  Repository.add_in_index r relpaths
```

8.2 Creating a blob from a file

Adding a set of files in the index consists simply in iterating over those files, modifying the index, and finally writing back the modified index on the disk:

⟨*signature* `Repository.add_in_index` 63a⟩≡ (179)
```
val add_in_index: t -> Fpath.t list -> unit
```

⟨*function* `Repository.add_in_index` 63b⟩≡ (180)
```
let add_in_index (r : t) (relpaths : Fpath.t list) : unit =
  ⟨Repository.add_in_index() sanity check relpaths 63c⟩
  relpaths |> List.iter (fun relpath ->
    ⟨Repository.add_in_index() adding relpath 63d⟩
  );
  write_index r
```
Uses `Repository.write_index()` 53b.

I described before the code of `Repository.write_index()`[53b] called above.

⟨`Repository.add_in_index()` *sanity check* `relpaths` 63c⟩≡ (63b)
```
assert (relpaths |> List.for_all (fun p -> Filename.is_relative !!p));
```

Adding a file in the index creates first a new blob object with the content of the added file. Then, `add_in_index()` adds the blob to the object store, and finally adds (or replaces) an entry in the index:

⟨`Repository.add_in_index()` *adding* `relpath` 63d⟩≡ (63b)
```
let full_path = r.worktree // relpath in
let stat =
  try Unix.lstat !!full_path
  with Unix.Unix_error _ ->
    failwith (spf "Repository.add_in_index: %s does not exist anymore"
                !!relpath)
in
let blob = Objects.Blob (content_from_path_and_unix_stat full_path stat) in
let sha = add_obj r blob in
let entry = Index.mk_entry relpath sha stat in
r.index <- Index.add_entry r.index entry;
```
Uses `Common.spf()`, `Index.add_entry()` 173b, `Index.mk_entry()` 173b, `Objects.t.Blob` 35d, `Repository./()`, `Repository.add_obj()` 49b, `Repository.content_from_path_and_unix_stat()` 64a, `Repository.t.index`, and `Repository.t.worktree` 35a.

I described before the general function `Repository.add_obj()`[49b] called above, and the specific code it uses to write a blob in `Blob.write()`[51a].

Note that the code above uses `Unix.lstat()`, and not `Unix.stat()` to deal with symbolic links (see Section 17.1 for more information on how Git handles symblic links).

Remember from Section 3.5 that the index contains sorted entries, so `Index.add_entry()` below must add the new entry at the right place:

⟨*signature* `Index.add_entry` 63e⟩≡ (173a)
```
val add_entry: t -> entry -> t
```

⟨*function* `Index.add_entry` 63f⟩≡ (173b)
```
let rec add_entry idx entry =
  match idx with
  | [] -> [entry]
  | x::xs ->
    (match entry.path <=> x.path with
    | Greater -> x::(add_entry xs entry)
    (* replacing old entry, new version of tracked file  *)
    | Equal -> entry::xs
    (* new file (the entries are sorted, no need to go through xs) *)
    | Less -> entry::x::xs
    )
```
Uses `Common.<=>()`, `Common.compare.Equal`, `Common.compare.Inf`, `Common.compare.Sup`, `Index.add_entry()` 173b, and `Index.entry.path` 39c.

The `<=>` operator used above is not a standard OCaml operator. It is inspired by a similar operator in Perl that compares values. It returns `Equal` if both values are equal, `Sup` if the first value is superior to the second one, or `Inf` if it is inferior. I show the code of `<=>` in Appendix D.

⟨*function* `Repository.content_from_path_and_unix_stat` 64a⟩≡ (180)
```
let content_from_path_and_unix_stat (full_path : Fpath.t) (stat : Unix.stats) : string =
  match stat.Unix.st_kind with
  ⟨Repository.content_from_path_and_unix_stat() match kind cases 132d⟩
  | Unix.S_REG ->
    full_path |> UChan.with_open_in (fun (ch : Chan.i) ->
      ch.ic |> IO.input_channel |> IO.read_all
    )
  | _ -> failwith (spf "Repository.add_in_index: %s kind not handled"
                       !!full_path)
```
Uses `Common.spf()`, `Common.with_file_in()`, `IO.input_channel()`, and `IO.read_all()`.

8.3 Removing files: `git rm`

⟨*constant* `Cmd_rm.cmd` 64b⟩≡ (156b)
```
let cmd = { Cmd_.
  name = "rm";
  usage = " [options] <file>...";
  options = [
    ⟨Cmd_rm.cmd command-line options ??⟩
  ];
  f = (fun args ->
    match args with
    | [] -> raise Cmd_.ShowUsage
    | xs ->
      let r, relpaths = Repository.find_root_open_and_adjust_paths (Fpath_.of_strings xs) in
      rm r relpaths
  );
}
```
Uses `Cmd.t.f` 55a, `Cmd.t.name` 55a, `Cmd.t.usage` 55a, `Cmd_rm.rm()` 64c, and `Repository.find_root_open_and_adjust_paths()` 61e.

The code for removing a file or set of files is even simpler. We just need to remove some entries from the index and write back the index. Indeed, as you will see in Chapter 9, committing what was staged consists mainly in creating a set of tree objects from the flat list of files in the index. If a file is not anymore in the index, then it will also not be part of any of those trees.

⟨*function* `Cmd_rm.rm` 64c⟩≡ (156b)
```
let rm r relpaths =
  (* less: not super efficient, could use hashes to speedup things *)
  r.Repository.index <-
    relpaths |> List.fold_left (fun idx relpath ->
          (* todo: -f? remove also file *)
      Index.remove_entry idx relpath
    ) r.Repository.index;
  Repository.write_index r
```
Uses `Index.remove_entry()` 173b and `Repository.t.index`.

`Index.remove_entry()` just does the opposite of `Index.add_entry()`[173b] in the previous section

⟨*signature* `Index.remove_entry` 64d⟩≡ (173a)
```
val remove_entry: t -> Fpath.t -> t
```

⟨*function* `Index.remove_entry` 65⟩≡ (173b)

```
let rec remove_entry idx (name : Fpath.t) =
  match idx with
  | [] -> failwith (spf "The file %s is not in the index" !!name)
  | x::xs ->
    (match name <=> x.path with
    | Greater -> x::(remove_entry xs name)
    | Equal -> xs
    (* the entries are sorted *)
    | Less -> failwith (spf "The file %s is not in the index" !!name)
    )
```

Uses `Common.<=>()`, `Common.compare.Equal`, `Common.compare.Inf`, `Common.compare.Sup`, `Common.spf()`, `Index.entry.path` 39c, and `Index.remove_entry()` 173b.

8.4 Renaming files: `git mv`

Chapter 9

Committing a Diff

In this chapter, you will see the second step of the commit process: the commit itself, which will use what was staged in the previous chapter.

9.1 Committing the index: `git commit`

Here is the code describing the `git commit` command:

⟨*constant* `Cmd_commit.cmd` 66a⟩≡ (154b)

```
let cmd = { Cmd_.
  name = "commit";
  usage = " [options]"; (* less: <pathspec>... *)
  options = [
    ⟨Cmd_commit.cmd command-line options ??⟩
  ];
  f = (fun args ->
    match args with
    | [] ->
      let r, _ = Repository.find_root_open_and_adjust_paths [] in
      ⟨Cmd_commit.cmd compute today 70c⟩
      ⟨Cmd_commit.cmd compute author user 70b⟩
      ⟨Cmd_commit.cmd compute committer user 70f⟩
      commit r author committer !message
    | _xs -> raise Cmd_.ShowUsage
  );
}
```

Uses `Cmd.t.f` 55a, `Cmd.t.name` 55a, `Cmd.t.options` 55a, `Cmd.t.usage` 55a, `Cmd_commit.commit()` 66b, `Cmd_commit.message` 70h, and `Repository.find_root_open_and_adjust_paths()` 61e.

`git commit` accepts a few flags from the command-line to specify the commit message or author. I will present those flags later in this chapter. What matters for now is that in the end `Cmd_commit.commit()` below will be called with the appropriate author, committer, and message.

⟨*function* `Cmd_commit.commit` 66b⟩≡ (154b)

```
let commit r author committer message =
  (* todo: imitate git output
   *   [master 0b50159] xxx
   *   1 file changed, 0 insertions(+), 0 deletions(-)
   *   create mode 100644 foobar.txt
   *)
  (* todo: nothing to commit, working directory clean *)
  Repository.commit_index r author committer message
```

Uses `Repository.commit_index()` 67b.

As I mentioned previously, committing a Diff consists mainly internally in transforming the current index in a set of tree objects as well as a commit object as shown in Figure **??**.

⟨*signature* `Repository.commit_index` 67a⟩≡ (179)
```
val commit_index:
  t -> User.t (* author *) -> User.t (* committer *) -> string (* msg *) -> unit
```

Most of the heavy work in the code below is done by `Index.trees_of_index()`[173b], which I will present in the next section. Some of the code below deals with commits that merge multiple branches, which I will describe later in Section 11.6.

⟨*function* `Repository.commit_index` 67b⟩≡ (180)
```
let commit_index r author committer message =
  let aref = Refs.Head in
  let root_tree = Index.trees_of_index r.index
    (fun t -> add_obj r (Objects.Tree t))
  in
  (* todo: execute pre-commit hook *)
  ⟨Repository.commit_index() read merge message if needed 81a⟩
  (* todo: execute commit-msg hook *)
  let commit = { Commit. parents = []; tree = root_tree;
                 author = author; committer = committer; message = message } in

  let ok =
    match follow_ref r aref |> snd with
    | None ->
      (* first commit so refs/heads/master does not even exist yet *)
      let sha = add_obj r (Objects.Commit commit) in
      ⟨Repository.commit_index() add ref when first commit 71a⟩
    | Some old_head ->
      ⟨Repository.commit_index() set merge_heads 81b⟩
      let commit = { commit with Commit.parents = old_head :: merge_heads } in
      let sha = add_obj r (Objects.Commit commit) in
      ⟨Repository.commit_index() update ref when not first commit 71b⟩
  in
  if not ok
  then failwith (spf "%s changed during commit" (Refs.string_of_ref aref));
  (* todo: execute post-commit hook *)
  ()
```
Uses `Commit.t.author` 36e, `Commit.t.committer` 37c, `Commit.t.message` 36e, `Commit.t.parents` 36e, `Commit.t.tree` 36e, `Common.spf()`, `Index.trees_of_index()` 173b, `Objects.t.Commit` 35d, `Objects.t.Tree` 35d, `Refs.string_of_ref()`, `Refs.t.Head` 37g, `Repository.add_obj()` 49b, `Repository.follow_ref()` 46b, and `Repository.t.index`.

9.2 Computing the trees from an index

The algorithm to transform an index in a set of trees operates in two steps:

1. The algorithm goes though all the files in the index to group them by directories while populating the local hashtable `dirs`
2. The algorithm builds tree objects by going recursively through the hashtable `dirs` and returns the SHA1 hash of the toplevel tree

Here is the type and skeleton of `Index.trees_of_index()`:

⟨*signature* `Index.trees_of_index` 67c⟩≡ (173a)
```
val trees_of_index: t -> (* add_obj *)(Tree.t -> Tree.hash) -> Tree.hash
```

⟨*function* `Index.trees_of_index` 68a⟩≡ (173b)
```
let trees_of_index idx add_tree_obj =
  let (dirs: dirs) = Hashtbl.create 11 in
  (* populate dirs *)
  ⟨Index.trees_of_index() populate dirs 68e⟩
  (* build trees *)
  ⟨Index.trees_of_index() build trees from dirs 69a⟩
```
Uses `Index.build_trees()` 173b.

9.2.1 Grouping index files by directories

Here is the type of the hashtbl `dirs` grouping files by directory:

⟨*type* `Index.dirs` 68b⟩≡ (173b)
```
type dirs = (Fpath.t (* full relpath of dir *), dir) Hashtbl.t
```

⟨*type* `Index.dir` 68c⟩≡ (173b)
```
type dir = dir_entry list ref
```

⟨*type* `Index.dir_entry` 68d⟩≡ (173b)
```
  and dir_entry =
    | Subdir of string (* basename *)
    | File of string (* basename *) * entry
```
Uses `Index.entry`.

Here is the code to populate the `dirs` hashtable:

⟨`Index.trees_of_index()` *populate* `dirs` 68e⟩≡ (68a)
```
Hashtbl.add dirs (Fpath.v ".") (ref []);
idx |> List.iter (fun entry ->
  let relpath : Fpath.t = entry.path in
  let (dirpath, base) = Filename.dirname !!relpath, Filename.basename !!relpath in
  let dir = find_dir dirs (Fpath.v dirpath) in
  dir := (File (base, entry))::!dir
);
```
Uses `Index.dir_entry.File` 68d, `Index.entry.path` 39c, and `Index.find_dir()` 173b.

Finding the directory of a file uses the function below:

⟨*function* `Index.add_dir` 68f⟩≡ (173b)
```
let rec find_dir (dirs : dirs) (dirpath : Fpath.t) : dir =
  try
    Hashtbl.find dirs dirpath
  with Not_found ->
    let newdir = ref [] in
    Hashtbl.add dirs dirpath newdir;
    ⟨Index.add_dir() recurse on parent of dirpath 68g⟩
    newdir
```

Note that adding a new directory in `dirs` requires also to recursively add its parent:

⟨`Index.add_dir()` *recurse on parent of* `dirpath` 68g⟩≡ (68f)
```
let (parent, base) =
  Filename.dirname !!dirpath, Filename.basename !!dirpath in
(* !recursive call! should stop at some point because "." is in dirs *)
let dir = find_dir dirs (Fpath.v parent) in
dir := Subdir (base)::!dir;
```
Uses `Index.dir_entry.Subdir` and `Index.find_dir()` 173b.

The recursion stops because `dirs` is initialized with an entry for `'.'` and `Filename.dirname` eventually returns `'.'` when it reaches the toplevel directory.

9.2.2 Building trees from directories

Here is the second step of the `trees_of_index()` algorithm, which builds a set of trees from `dirs` recursively starting from the root directory `'.'`.

⟨`Index.trees_of_index()` *build trees from* `dirs` 69a⟩≡ (68a)
```
build_trees dirs (Fpath.v ".") add_tree_obj
```

⟨*function* `Index.build_trees` 69b⟩≡ (173b)
```
let rec build_trees (dirs : dirs) (dirpath : Fpath.t) add_tree_obj =
  let dir = Hashtbl.find dirs dirpath in
  (* entries of a Tree.t must be sorted, but entries of an index too,
   * so we can assume add_dir was called in sorted order
   *)
  let xs = List.rev !dir in
  let tree =
    xs |> List.map (function
      | File (base, entry) ->
        {Tree.
         name = base;
         id = entry.id;
         perm = perm_of_mode entry.stats.mode;
        }
      | Subdir base ->
        (* recurse *)
        let sha =
          build_trees dirs (dirpath / base) add_tree_obj in
        {Tree. perm = Tree.Dir; name = base; id = sha }
    )
  in
  add_tree_obj tree
```

Uses `Index.build_trees()` 173b, `Index.dir_entry.File` 68d, `Index.dir_entry.Subdir`, `Index.entry.id` 39c, `Index.entry.stats` 39c, `Index.perm_of_mode()` 173b, `Index.stat_info.mode` 39d, `Tree.entry.id` 36b, `Tree.entry.name` 36b, `Tree.entry.perm` 36b, and `Tree.perm.Dir` 36d.

The function `add_tree_obj()` above will not only add a tree object the repository. It will also return its SHA1. Its value (`fun t -> add_obj (Tree t)`) is passed down to `build_trees` from `Repository.commit_index()`[67b]

The code above relies on the helper function below to convert an `Index.mode`[39e] to a `Tree.perm`[36d]:

⟨*signature* `Index.perm_of_mode` 69c⟩≡ (173a)
```
val perm_of_mode: mode -> Tree.perm
```

⟨*function* `Index.perm_of_mode` 69d⟩≡ (173b)
```
let perm_of_mode mode =
  match mode with
  | Normal -> Tree.Normal
  | Exec -> Tree.Exec
  | Link -> Tree.Link
  ⟨Index.perm_of_mode() match mode cases 133d⟩
```

Uses `Index.mode.Exec` 39e, `Index.mode.Gitlink` 132b, `Index.mode.Link` 39e, `Index.mode.Normal` 39e, `Tree.perm.Commit` 132a, `Tree.perm.Exec` 36d, `Tree.perm.Link` 36d, and `Tree.perm.Normal` 36d.

9.3 Storing the date, author, and committer

Once `Repository.commit_index()`[67b] computed the toplevel tree object from the index, setting the other fields of the commit object is trivial.

⟨`Cmd_commit.cmd` *command-line options* 69e⟩+≡ (66a) ◁ ?? 70d ▷
```
"--author",    Arg.Set_string author, " <author>";
```

Uses `Cmd_commit.author` 70a.

⟨*constant* `Cmd_commit.author` 70a⟩≡ (154b)
```
let author = ref ""
```

⟨`Cmd_commit.cmd` *compute* `author` *user* 70b⟩≡ (66a)
```
(* todo: read from .git/config or ~/.gitconfig *)
let author =
  if !author = ""
  then { User.
         name = Unix.getlogin ();
         email = "todo@todo";
         date = today;
       }
  else raise Todo (* need parse author string *)
in
```
Uses `User.t.date` 37a, `User.t.email` 37a, and `User.t.name` 37a.

⟨`Cmd_commit.cmd` *compute* `today` 70c⟩≡ (66a)
```
let today =
  (Int64.of_float (Unix.time ()),
   (* todo: use localtime vs gmtime? *) -7 (* SF *))
in
```

⟨`Cmd_commit.cmd` *command-line options* 70d⟩+≡ (66a) ◁69e 70g▷
```
"--committer", Arg.Set_string committer, " <author>";
```
Uses `Cmd_commit.committer` 70e.

⟨*constant* `Cmd_commit.committer` 70e⟩≡ (154b)
```
let committer = ref ""
```

⟨`Cmd_commit.cmd` *compute* `committer` *user* 70f⟩≡ (66a)
```
let committer =
  if !committer = ""
  then author
  else raise Todo
in
```

9.4 Recording the message

⟨`Cmd_commit.cmd` *command-line options* 70g⟩+≡ (66a) ◁70d
```
"-m",        Arg.Set_string message, " <msg> commit message";
"--message", Arg.Set_string message, " <msg> commit message";
```
Uses `Cmd_commit.message` 70h.

⟨*constant* `Cmd_commit.message` 70h⟩≡ (154b)
```
let message = ref ""
```

9.5 Updating `HEAD` atomically

The last operation of `Repository.commit_index()`[67b] is to update `HEAD` to point to the newly created commit object. In fact, `git commit` does not update `HEAD` itself but the head pointed by `HEAD` (e.g., `.git/refs/heads/master`), hence the use of `Repository.follow_ref()`[46b] in the code of the next sections. Updating this head must be done atomically. To do so, `Repository.commit_index()` rely on two different functions to handle two different situations:

1. When this is the first commit and `HEAD` points to an head that does not exist yet. In that case it calls `Repository.add_ref_if_new()`[71d].

2. When there was already an older previous commit. In that case it calls `Repository.set_ref_if_same_old()`[72].

⟨`Repository.commit_index()` *add ref when first commit* 71a⟩≡ (67b)
```
add_ref_if_new r aref (Refs.Hash sha)
```
Uses `Refs.ref_content.Hash` 38b and `Repository.add_ref_if_new()` 71d.

⟨`Repository.commit_index()` *update ref when not first commit* 71b⟩≡ (67b)
```
set_ref_if_same_old r aref old_head sha
```
Uses `Repository.set_ref_if_same_old()` 72.

Both functions above return a boolean indicating whether the operation could be done atomically. For example, `Repository.set_ref_if_same_old()` will return false if while modifying the reference, another process modified it before and the current value is not anymore the old one read before.

9.5.1 Creating a new reference (first commit)

Creating the first commit and the first head atomically requires to make sure that while creating the reference file, nobody else created the same file.

⟨*signature* `Repository.add_ref_if_new` 71c⟩≡ (179)
```
val add_ref_if_new: t -> Refs.t -> Refs.ref_content -> bool
```

⟨*function* `Repository.add_ref_if_new` 71d⟩≡ (180)
```
let add_ref_if_new r aref refval =
  let (refs, shaopt) = follow_ref r aref in
  if shaopt <> None
  then false
  else begin
    let lastref = List.hd (List.rev refs) in
    let file = ref_to_filename r lastref in
    (* todo: ensure dirname exists *)
    file |> with_file_out_with_lock (fun ch ->
      (* todo: check file does not exist aleady; then return false! *)
      ch |> IO.output_channel |> IO_.with_close_out (Refs.write refval);
      true
    )
  end
```
Uses `IO.output_channel()`, `IO_.with_close_out()` 146c, `Refs.write()`, `Repository.follow_ref()` 46b, `Repository.ref_to_filename()` 38e, and `Repository.with_file_out_with_lock()` 50d.

5.1.2
I described the code of `Refs.write()`[52i] before.

9.5.2 Updating an existing reference

⟨*signature* `Repository.set_ref_if_same_old` 71e⟩≡ (179)
```
val set_ref_if_same_old: t -> Refs.t -> Sha1.t -> Sha1.t -> bool
```

⟨*function* `Repository.set_ref_if_same_old` 72⟩≡ (180)

```
let set_ref_if_same_old r aref _oldh newh =
  let (refs, _) = follow_ref r aref in
  let lastref = List.hd (List.rev refs) in
  let file = ref_to_filename r lastref in
  try
    file |> with_file_out_with_lock (fun ch ->
      (* TODO generate some IO.No_more_input
      let prev = read_ref r lastref in
      if prev <> (Refs.Hash oldh)
      then raise Not_found
      else
      *)
        ch |> IO.output_channel |> IO_.with_close_out
            (Refs.write (Refs.Hash newh))
    );
    true
  with Not_found -> false
```

Uses `IO.output_channel()`, `IO_.with_close_out()` 146c, `Refs.ref_content.Hash` 38b, `Refs.write()`, `Repository.follow_ref()` 46b, `Repository.ref_to_filename()` 38e, and `Repository.with_file_out_with_lock()` 50d.

Chapter 10

Branching

In this chapter, you will see the code to manipulate branches. Git uses the same command, `git branch`, whose code is below, to list, create, and delete branches depending on the command-line arguments. You will also see in this chapter the code of `git checkout` to switch between branches, and `git reset` to reset all modifications done while working on the current branch.

⟨*constant* `Cmd_branch.cmd` 73a⟩≡ (153c)
```
let cmd = { Cmd_.
  name = "branch";
  usage = " [options]
   or: ocamlgit branch [options] <branchname>
   or: ocamlgit branch [options] (-d | -D) <branchname>
";
  options = [
    ⟨Cmd_branch.cmd command-line options ??⟩
  ];
  f = (fun args ->
    let r, _ = Repository.find_root_open_and_adjust_paths [] in
    match args with
    ⟨Cmd_branch.cmd match args cases ??⟩
    | _ -> raise Cmd_.ShowUsage
  );
}
```
Uses `Cmd.t.options` 55a, `Cmd.t.usage` 55a, and `Repository.find_root_open_and_adjust_paths()` 61e.

10.1 Listing branches: `git branch`

To list the set of active branches, you just need to call `git branch` without any argument:

⟨`Cmd_branch.cmd` *match args cases* 73b⟩+≡ (73a) ◁?? 74c▷
```
| [] -> list_branches r
```

Here is the output of `git branch` on a repository with only two branches. `git branch` marks the current branch with a `*` before its name.

```
1 $ git branch
  experiment
* master
```

Here is the code of `list_branches()`:

⟨*function* `Cmd_branch.list_branches` 73c⟩≡ (153c)
```
(* less: remote_flag set with --all to also list remote refs *)
```

```
let list_branches r =
  let head_branch = Repository.read_ref r (Refs.Head) in
  let all_refs = Repository.all_refs r in
  all_refs |> List.iter (fun refname ->
    if refname =~ "^refs/heads/\\(.*\\)"
    then
      let short = Regexp_.matched1 refname in
      let prefix =
        if (Refs.OtherRef refname = head_branch)
        then "* "
        else "  "
      in
      UConsole.print (spf "%s%s" prefix short)
  )
```

Uses Common.Regexp_.matched1(), Common.pr(), Common.spf(), Refs.ref_content.OtherRef 38b, Refs.t.Head 37g, Repository.all_refs() 74b, and Repository.read_ref() 45e.

I described before the code of `Repository.read_ref()`[45e]. Thanks to the simplicity of how Git stores references on the disk, getting all the references consists just in walking the `.git/refs/` directory:

⟨*signature* Repository.all_refs 74a⟩≡ (179)

```
val all_refs: t -> Refs.refname list
```

⟨*function* Repository.all_refs 74b⟩≡ (180)

```
let all_refs (r : t) : Refs.refname list =
  let root = r.dotgit in
  let rootlen = String.length !!root in
  let res = ref [] in
  (root / "refs") |> walk_dir (fun path _dirs files ->
    files |> List.iter (fun file ->
      (* less: replace os.path.sep *)
      let dir = String.sub !!path rootlen (String.length !!path - rootlen) in
      let refname = Fpath.v dir / file in
      Stack_.push !!refname res
    );
   );
  List.rev !res
```

Uses Common.push(), Repository./(), Repository.t.dotgit 35a, and Repository.walk_dir() 145b.

The code above relies on the utility function `Repository.walk_dir()`[145b] to explore recursively a directory (see Appendix D for its code). This function takes a callback function and a directory as parameters and calls the callback for each subdirectories of the directory parameter. The callback is called with three arguments: the path of the current subdirectory, the immediate subdirectories in this subdirectory, and the immediate files in this subdirectory.

10.2 Creating a branch: `git branch <name>`

To create a new branch, you need to call `git branch` with one argument: the name of the desired branch.

⟨Cmd_branch.cmd *match args cases* 74c⟩+≡ (73a) ◁73b

```
| [name] ->
  (match () with
  ⟨Cmd_branch.cmd when one argument, when flags cases 75e⟩
  | _ -> create_branch r name
  )
```

Uses Cmd_branch.delete_branch() 153c.

Creating a branch simply creates a new reference with the same SHA1 than the SHA1 of the last commit in the current branch:

⟨*function* `Cmd_branch.create_branch` 75a⟩≡ (153c)
```
let create_branch r name (* sha *) =
  let refname = "refs/heads/" ^ name in
  ⟨Cmd_branch.create_branch() sanity check refname 75b⟩
  let sha = Repository.follow_ref_some r (Refs.Head) in
  let ok = Repository.add_ref_if_new r (Refs.Ref refname) (Refs.Hash sha) in
  if not ok
  then failwith (spf "could not create branch '%s'" name)
```
Uses `Common.spf()` and `Repository.all_refs()` 74b.

⟨`Cmd_branch.create_branch()` *sanity check refname* 75b⟩≡ (75a)
```
let all_refs = Repository.all_refs r in
if List.mem refname all_refs
(* less: unless -force *)
then failwith (spf "A branch named '%s' already exists." name);
```
Uses `Common.spf()`, `Refs.t.Head` 37g, and `Repository.follow_ref_some()` 46d.

10.3 Deleting a branch: `git branch -d <name>`

Finally, to delete a branch, adds the `-d` (or `-D`) command-line flag before the name of the branch:

⟨`Cmd_branch.cmd` *command-line options* 75c⟩+≡ (73a) ◁?? 75g▷
```
"-d",       Arg.Set del_flag, " delete fully merged branch";
"--delete", Arg.Set del_flag, " delete fully merged branch";
```
Uses `Cmd_branch.del_force`.

⟨*constant* `Cmd_branch.del_flag` 75d⟩≡ (153c)
```
let del_flag = ref false
```

⟨`Cmd_branch.cmd` *when one argument, when flags cases* 75e⟩≡ (74c) 75f▷
```
| _ when !del_flag  -> delete_branch r name false
```
Uses `Cmd_branch.delete_branch()` 153c.

⟨`Cmd_branch.cmd` *when one argument, when flags cases* 75f⟩+≡ (74c) ◁75e
```
| _ when !del_force -> delete_branch r name true
```
Uses `Cmd_branch.create_branch()` 153c.

⟨`Cmd_branch.cmd` *command-line options* 75g⟩+≡ (73a) ◁75c
```
"-D",       Arg.Set del_force, " delete branch (even if not merged)";
```

⟨*constant* `Cmd_branch.del_force` 75h⟩≡ (153c)
```
let del_force = ref false
```

⟨*function* `Cmd_branch.delete_branch` 75i⟩≡ (153c)
```
let delete_branch r name force =
  let refname = "refs/heads/" ^ name in
  let aref = Refs.Ref refname in
  let sha = Repository.follow_ref_some r aref in
  ⟨Cmd_branch.delete_branch() sanity check if branch merged unless force 76b⟩
  Repository.del_ref r aref;
  UConsole.print (spf "Deleted branch %s (was %s)" name (Hexsha.of_sha sha))
```
Uses `Refs.t.Ref` 37g and `Repository.follow_ref_some()` 46d.

Deleting a branch consists simply in deleting a reference file under `.git/refs`:

⟨*signature* `Repository.del_ref` 75j⟩≡ (179)
```
val del_ref: t -> Refs.t -> unit
```

⟨*function* `Repository.del_ref` 76a⟩≡ (180)
```
let del_ref r aref =
  let file = ref_to_filename r aref in
  Unix.unlink !!file
```
Uses `Repository.ref_to_filename()` 38e.

⟨`Cmd_branch.delete_branch()` *sanity check if branch merged unless force* 76b⟩≡ (75i)
```
if not force
(* todo: detect if fully merged branch! *)
then ();
```
Uses `Repository.del_ref()` 76a.

10.4 Checking out a branch: `git checkout`

To switch to another branch, Git uses another command: `git checkout`.

⟨*constant* `Cmd_checkout.cmd` 76c⟩≡ (153d)
```
let cmd = { Cmd_.
  name = "checkout";
  usage = " [options] <branch>
   or: ocamlgit checkout [options] <commitid>
   or: ocamlgit checkout [options]
";
  options = [
   ⟨Cmd_checkout.cmd command-line options ??⟩
  ];
  f = (fun args ->
    let r, _ = Repository.find_root_open_and_adjust_paths [] in
    match args with
    ⟨Cmd_checkout.cmd match args cases 76d⟩
    | _ -> raise Cmd_.ShowUsage
  );
}
```
Uses `Cmd.t.f` 55a, `Cmd.t.name` 55a, `Cmd.t.options` 55a, `Cmd.t.usage` 55a, and `Repository.find_root_open_and_adjust_paths()` 61e.

This command has also multiple operation modes depending on the its command-line arguments. With an argument, `git checkout` will switch the current branch:

⟨`Cmd_checkout.cmd` *match args cases* 76d⟩≡ (76c) 77a▷
```
| [str] -> checkout r str
```
Uses `Cmd_checkout.checkout()` 76e.

This argument can be the name of a branch (e.g., `git checkout experiment`), but also a string denotting a past commit (e.g., `git checkout 19d977`). Here is the code below to deal with a branch name as an argument:

⟨*function* `Cmd_checkout.checkout` 76e⟩≡ (153d)
```
let checkout r str =
  let all_refs = Repository.all_refs r in
  let refname = "refs/heads/" ^ str in

  match () with
  | _ when List.mem refname all_refs ->
    let commitid = Repository.follow_ref_some r (Refs.Ref refname) in
    let commit = Repository.read_commit r commitid in
    let tree = Repository.read_tree r commit.Commit.tree in

    (* todo: order of operation? set ref before index? reverse? *)
    Repository.write_ref r (Refs.Head) (Refs.OtherRef refname);
```

```
    Repository.set_worktree_and_index_to_tree r tree;
    UConsole.print (spf "Switched to branch '%s'" str);
    (* less: if master, then check if up-to-date with origin/master *)
```

⟨Cmd_checkout.checkout() *cases* 97⟩
```
| _ -> raise Cmd_.ShowUsage
```

Uses Commit.t.tree 36e, Common.pr(), Common.spf(), Refs.ref_content.OtherRef 38b, Refs.t.Head 37g, Refs.t.Ref 37g, Repository.all_refs() 74b, Repository.follow_ref_some() 46d, Repository.read_commit() 45c, Repository.read_tree() 43f, Repository.set_worktree_and_index_to_tree() 77d, and Repository.write_ref() 52g.

To checkout a branch, the code above first finds the commit associated with the reference of the branch, then reads it from the disk (see the code of `Repository.read_commit()`[45c]), and then read its tree (see the code of `Repository.read_tree()`[43f]). Then, most of the heavy work is done by `Repository.set_worktree_and_index_to_` which I will present in the next section.

⟨Cmd_checkout.cmd *match args cases* 77a⟩+≡ (76c) ◁76d
```
| [] -> update r
```

Uses Cmd_checkout.update() 77b.

⟨*function* Cmd_checkout.update 77b⟩≡ (153d)
```
(* Your branch is up-to-date with 'origin/master'. *)
let update _r =
  raise Todo
```

10.4.1 Computing the index (and work tree) from trees

⟨*signature* Repository.set_worktree_and_index_to_tree 77c⟩≡ (179)
```
val set_worktree_and_index_to_tree:
  t -> Tree.t -> unit
```

⟨*function* Repository.set_worktree_and_index_to_tree 77d⟩≡ (180)
```
let set_worktree_and_index_to_tree r tree =
  (* todo: need lock on index? on worktree? *)
  let hcurrent =
    r.index |> List.map (fun e -> e.Index.path, false) |> Hashtbl_.of_list in
  let new_index = ref [] in
  (* less: honor file mode from config file? *)
  tree |> Tree.walk_tree (read_tree r) (Fpath.v "XXX") (fun relpath entry ->
    let perm = entry.Tree.perm in
    match perm with
    | Tree.Dir ->
      (* bugfix: need also here to mkdir; doing it below is not enough
       * when a dir has no file but only subdirs
       *)
      let fullpath = r.worktree // relpath in
      if not (Sys.file_exists !!fullpath)
      then Unix.mkdir !!fullpath dirperm;
    | Tree.Normal | Tree.Exec | Tree.Link ->
      (* less: validate_path? *)
      let fullpath = r.worktree // relpath in
      if not (Sys.file_exists (Filename.dirname !!fullpath))
      then Unix.mkdir (Filename.dirname !!fullpath) dirperm;
      let sha = entry.Tree.id in
      let blob = read_blob r sha in
      let stat = build_file_from_blob fullpath blob perm in
      Hashtbl.replace hcurrent relpath true;
      Stack_.push (Index.mk_entry relpath sha stat) new_index;
    ⟨Repository.set_worktree_and_index_to_tree() walk tree cases 132h⟩
  );
```

```
  let index = List.rev !new_index in
  r.index <- index;
  write_index r;
  hcurrent |> Hashtbl.iter (fun file used ->
    if not used
    then
      (* todo: should check if modified? otherwise lose modif! *)
      let fullpath = r.worktree // file in
      Unix.unlink !!fullpath
  )
  (* less: delete if a dir became empty, just walk_dir? *)
```

Uses Common.Hashtbl_.of_list(), Common.push(), Index.entry.path 39c, Index.mk_entry() 173b, Repository./(), Repository.build_file_from_blob() 78c, Repository.dirperm 60a, Repository.read_blob() 42g, Repository.read_tree() 43f, Repository.t.index, Repository.t.worktree 35a, Repository.write_index() 53b, Tree.entry.id 36b, Tree.entry.perm 36b, Tree.perm.Dir 36d, Tree.perm.Exec 36d, Tree.perm.Link 36d, Tree.perm.Normal 36d, and Tree.walk_tree() 78b.

⟨*signature* Tree.walk_tree 78a⟩≡ (183a)

```
val walk_tree:
  (hash -> t) -> Fpath.t (* dir *) ->
  (Fpath.t -> entry -> unit) -> t -> unit
```

⟨*function* Tree.walk_tree 78b⟩≡ (183b)

```
(* we must visit in sorted order, so the caller of walk_tree can rely on 'f'
 * being called in order (so it can easily create for example sorted
 * index entries while visiting a tree)
 *)
let rec walk_tree read_tree (dirpath : Fpath.t) f xs =
  xs |> List.iter (fun entry ->
    let relpath = dirpath / entry.name in
    f relpath entry;
    match entry.perm with
    | Dir ->
      walk_tree read_tree relpath f (read_tree entry.id)
    | Normal | Exec | Link -> ()
    ⟨Tree.walk_tree() match perm cases 132i⟩
  )
```

Uses Tree.entry.id 36b, Tree.entry.name 36b, Tree.entry.perm 36b, Tree.perm.Commit 132a, Tree.perm.Dir 36d, Tree.perm.Exec 36d, Tree.perm.Link 36d, Tree.perm.Normal 36d, and Tree.walk_tree() 78b.

10.4.2 Creating a file from a blob

⟨*function* Repository.build_file_from_blob 78c⟩≡ (180)

```
let build_file_from_blob (fullpath : Fpath.t) blob perm =
  let oldstat =
    try
      Some (Unix.lstat !!fullpath)
    with Unix.Unix_error _ -> None
  in
  (match perm with
  | Tree.Link ->
    if oldstat <> None
    then Unix.unlink !!fullpath;
    Unix.symlink blob !!fullpath;
  | Tree.Normal | Tree.Exec ->
    (match oldstat with
    (* opti: if same content, no need to write anything *)
    | Some { Unix.st_size = x; _ } when x = String.length blob &&
```

```
      (fullpath |> UChan.with_open_in (fun (ch : Chan.i) ->
        (ch.ic |> IO.input_channel |> IO.read_all ) = blob
       )) ->
      ()
    | _ ->
      fullpath |> UChan.with_open_out (fun (ch : Chan.o) ->
        output_bytes ch.oc (Bytes.of_string blob)
      );
      (* less: honor filemode? *)
      Unix.chmod !!fullpath
        (match perm with
        | Tree.Normal -> 0o644
        | Tree.Exec -> 0o755
        | _ -> raise (Impossible "matched before")
        )
  )
| Tree.Dir -> raise (Impossible "dirs filtered in walk_tree iteration")
⟨Repository.build_file_from_blob() match perm cases 132j⟩
);
Unix.lstat !!fullpath
```

Uses Common.with_file_out(), Tree.perm.Dir 36d, Tree.perm.Exec 36d, Tree.perm.Link 36d, and Tree.perm.Normal 36d.

10.5 Resetting a branch: `git reset`

Resetting a branch is useful for example you want to quickly undo the modifications you have done to your working files. To do so you must use the `git reset` command with the `-hard` command-line flag. Indeed, there are multiple ways to reset a branch, as shown by the code and flags below:

⟨*constant* Cmd_reset.cmd 79a⟩≡ (156a)

```
let cmd = { Cmd_.
  name = "reset";
  usage = " [options] ";
  options = [
    ⟨Cmd_reset.cmd command-line options ??⟩
  ];
  f = (fun args ->
    let r, _ = Repository.find_root_open_and_adjust_paths [] in
    match args with
    | [] ->
      (match () with
      ⟨Cmd_reset.cmd.f() when no args, cases 80c⟩
      | _ -> raise Cmd_.ShowUsage
      )
    | _ -> raise Cmd_.ShowUsage
  );
}
```

Uses Cmd.t.f 55a, Cmd.t.name 55a, Cmd.t.options 55a, Cmd.t.usage 55a, and Repository.find_root_open_and_adjust_paths() 61e.

⟨Cmd_reset.cmd *command-line options* 79b⟩+≡ (79a) ◁??

```
"--hard", Arg.Set hard, " reset HEAD, index and working tree";
"--soft", Arg.Set soft, " reset only HEAD";
"--mixed", Arg.Set mixed, " reset HEAD and index";
```

Uses Cmd_reset.hard 79c, Cmd_reset.mixed 80b, and Cmd_reset.soft 80a.

⟨*constant* Cmd_reset.hard 79c⟩≡ (156a)

```
let hard = ref false
```

⟨*constant* `Cmd_reset.soft` 80a⟩≡ (156a)
```
let soft = ref false
```

⟨*constant* `Cmd_reset.mixed` 80b⟩≡ (156a)
```
let mixed = ref false
```

⟨`Cmd_reset.cmd.f()` *when no args, cases* 80c⟩≡ (79a)
```
| _ when !hard -> reset_hard r
```
Uses `Cmd_reset.reset_hard()` 80d.

A hard reset is equivalent to checking out again the commit of the current branch, and so the code below can reuse `Repository.set_worktree_and_index_to_tree()`[77d] used for the code of `git checkout`.

⟨*function* `Cmd_reset.reset_hard` 80d⟩≡ (156a)
```
let reset_hard r =
  let commitid = Repository.follow_ref_some r (Refs.Head) in
  let commit = Repository.read_commit r commitid in
  let tree = Repository.read_tree r commit.Commit.tree in

  Repository.set_worktree_and_index_to_tree r tree;
  UConsole.print (spf "HEAD is now at %s %s"
        (String.sub (Hexsha.of_sha commitid) 0 6)
        (String.sub commit.Commit.message 0 40))
```
Uses `Commit.t.message` 36e, `Commit.t.tree` 36e, `Common.pr()`, `Common.spf()`, `Hexsha.of_sha()`, `Refs.t.Head` 37g, `Repository.follow_ref_some()` 46d, `Repository.read_commit()` 45c, `Repository.read_tree()` 43f, and `Repository.set_worktree_and_index_to_tree()` 77d.

⟨`Cmd_reset.cmd.f()` *if not hard reset* 80e⟩≡
```
if !soft || !mixed || not !hard
then begin
  pr2 "only --hard supported";
  raise Cmd_.ShowUsage
end
```

Chapter 11

Merging

11.1 merging branches: `git merge`

11.2 Merging two trees

11.3 Merging two files

11.4 `diff3` and `merge`

11.5 Merging conflicts

11.6 Committing merged branches: `git commit`

⟨`Repository.commit_index()` *read merge message if needed* 81a⟩≡ (67b)
```
(* less: Try to read commit message from .git/MERGE_MSG *)
```

⟨`Repository.commit_index()` *set* `merge_heads` 81b⟩≡ (67b)
```
(* less: merge_heads from .git/MERGE_HEADS *)
let merge_heads = [] in
```

Chapter 12

Inspecting

12.1 Showing the content of an object: `git show`

⟨*constant* `Cmd_show.cmd` 82a⟩≡ (156c)
```
let cmd = { Cmd_.
  name = "show";
  usage = " <objectish>";
  (* less: --oneline *)
  options = [];
  f = (fun args ->
    let r, _ = Repository.find_root_open_and_adjust_paths [] in
    match args with
    | [] -> show r (Repository.ObjByRef (Refs.Head))
    | xs ->
      xs |> List.iter (fun str ->
        show r (Repository.parse_objectish str)
      )
  );
}
```
Uses `Cmd.t.f` 55a, `Cmd.t.name` 55a, `Cmd.t.options` 55a, `Cmd.t.usage` 55a, `Cmd_show.show()` 82b, `Refs.t.Head` 37g, `Repository.find_root_open_and_adjust_paths()` 61e, `Repository.objectish.ObjByRef` 38f, and `Repository.parse_objectish()` 180.

⟨*function* `Cmd_show.show` 82b⟩≡ (156c)
```
let show r objectish =
  let sha, obj = Repository.read_objectish r objectish in
  match obj with
  ⟨Cmd_show.show() match obj cases 82c⟩
```
Uses `Repository.read_objectish()` 46f.

12.1.1 Showing a blob

⟨`Cmd_show.show()` *match obj cases* 82c⟩≡ (82b) 83a▷
```
| Objects.Blob x ->
  Blob.show x
```
Uses `Blob.show()` 82e and `Objects.t.Blob` 35d.

⟨*signature* `Blob.show` 82d⟩≡ (148c)
```
val show: t -> unit
```

⟨*function* `Blob.show` 82e⟩≡ (149a)
```
let show x =
  print_string x
```

12.1.2 Showing a tree

⟨Cmd_show.show() *match obj cases* 83a⟩+≡ (82b) ◁82c 83d▷
```
| Objects.Tree x ->
  (* =~ git ls-tree --names-only *)
  UConsole.print (spf "tree %s\n" (Hexsha.of_sha sha));
  Tree.show x
```
Uses Common.pr(), Common.spf(), Hexsha.of_sha(), Objects.t.Tree 35d, and Tree.show() 183b.

⟨*signature* Tree.show 83b⟩≡ (183a)
```
val show: t -> unit
```

⟨*function* Tree.show 83c⟩≡ (183b)
```
let show xs =
  xs |> List.iter (fun entry ->
    UConsole.print (spf "%s%s" entry.name
          (match entry.perm with
          | Dir -> "/"
          | _ -> ""
          ))
  )
```
Uses Common.pr(), Common.spf(), Tree.entry.name 36b, Tree.entry.perm 36b, and Tree.perm.Dir 36d.

12.1.3 Showing a commit

⟨Cmd_show.show() *match obj cases* 83d⟩+≡ (82b) ◁83a 134f▷
```
| Objects.Commit x ->
  UConsole.print (spf "commit %s" (Hexsha.of_sha sha));
  Commit.show x;
  let tree2 = Repository.read_tree r x.Commit.tree in
  let tree1 =
    try
      let parent1 = Repository.read_commit r (List.hd x.Commit.parents) in
      Repository.read_tree r parent1.Commit.tree
    with Failure _ ->
    (* no parent *)
      []
  in
  let changes =
    Changes.changes_tree_vs_tree
      (Repository.read_tree r)
      (Repository.read_blob r)
      tree1 tree2
  in
  changes |> List.iter Diff_unified.show_change
```
Uses Changes.changes_tree_vs_tree() 91c, Commit.show() 83f, Commit.t.parents 36e, Commit.t.tree 36e, Common.pr(), Common.spf(), Diff_unified.show_change(), Hexsha.of_sha(), Objects.t.Commit 35d, Repository.read_blob() 42g, Repository.read_commit() 45c, and Repository.read_tree() 43f.

⟨*signature* Commit.show 83e⟩≡ (159a)
```
val show: t -> unit
```

⟨*function* Commit.show 83f⟩≡ (159b)
```
let show x =
  UConsole.print (spf "Author: %s <%s>" x.author.User.name x.author.User.email);
  (* less: date of author or committer? *)
  let date = x.author.User.date in
  UConsole.print (spf "Date:   %s" (User.string_of_date date));
  UConsole.print "";
```

```
    UConsole.print ("    " ^ x.message)
    (* showing diff done in caller in Cmd_show.show *)
```

Uses Commit.t.author 36e, Commit.t.message 36e, Common.pr(), Common.spf(), User.string_of_date() 84b, User.t.date 37a, User.t.email 37a, and User.t.name 37a.

⟨*signature* User.string_of_date 84a⟩≡ (187a)

```
  (* for show *)
  val string_of_date: (int64 * timezone_offset) -> string
```

⟨*function* User.string_of_date 84b⟩≡ (187b)

```
  let string_of_date (date, tz) =
    let f = Int64.to_float date in
    let tm = Unix.localtime f in

    spf "%s %s %d %02d:%02d:%02d %d %c%02d%02d"
      (Date.string_of_day tm.Unix.tm_wday) (Date.string_of_month tm.Unix.tm_mon)
      tm.Unix.tm_mday
      tm.Unix.tm_hour tm.Unix.tm_min tm.Unix.tm_sec (tm.Unix.tm_year + 1900)
      (char_of_sign tz) (abs tz) 0
```

Uses Common.spf(), Date.string_of_day(), Date.string_of_month(), and User.char_of_sign() 84c.

⟨*function* User.char_of_sign 84c⟩≡ (187b)

```
  let char_of_sign = function
    | x when x >= 0 -> '+'
    | _ -> '-'
```

12.2 Representing changes

12.2.1 Tree changes

⟨*type* Change.t 84d⟩≡ (149)

```
  (* entry below refers only to files (not dirs), and their name
   * are adjusted to show a relative path from the root of the
   * project.
   *)
  type t =
    | Add of entry
    | Del of entry
    | Modify of entry * entry (* before / after *)
    (* less: Rename, Copy *)
    (*| Identical of Tree.entry *)
```

Uses Change.entry 84e.

⟨*type* Change.entry 84e⟩≡ (149)

```
  type entry = {
    (* relative path *)
    path: Fpath.t;
    mode: Index.mode;
    content: content Lazy.t;
  }
```

Uses Index.mode.

⟨*type* Change.content 84f⟩≡ (149)

```
  type content = string
```

12.2.2 File changes

⟨*type* `Diff.diff` 85a⟩≡ (161a 160c)
```
type diff = (item diff_elem) list
```

⟨*type* `Diff.diff_elem` 85b⟩≡ (161a 160c)
```
(* similar to change.ml, but for content of the file *)
type 'item diff_elem =
  | Added   of 'item
  | Deleted of 'item
  | Equal   of 'item
```

⟨*type* `Diff.item` 85c⟩≡ (161a 160c)
```
type item = string
```

12.3 Showing file differences: `git diff`

⟨*constant* `Cmd_diff.cmd` 85d⟩≡ (154c)
```
let cmd = { Cmd_.
  name = "diff";
  usage = " ";
  options = [];
  f = (fun args ->
    let r, _ = Repository.find_root_open_and_adjust_paths [] in
    match args with
    | [] -> diff_worktree_vs_index r
    | _xs -> raise Cmd_.ShowUsage
  );
}
```

Uses `Cmd.t.f` 55a, `Cmd.t.name` 55a, `Cmd.t.options` 55a, `Cmd.t.usage` 55a, `Cmd_diff.diff_worktree_vs_index()` 85e, and `Repository.find_root_open_and_adjust_paths()` 61e.

⟨*function* `Cmd_diff.diff_worktree_vs_index` 85e⟩≡ (154c)
```
let diff_worktree_vs_index r =
  let changes =
    Changes.changes_worktree_vs_index
      (Repository.read_blob r)
      r.Repository.worktree
      r.Repository.index
  in
  changes |> List.iter Diff_unified.show_change
```

Uses `Changes.changes_worktree_vs_index()` 86a, `Repository.read_blob()` 42g, `Repository.t.index`, and `Repository.t.worktree` 35a.

12.3.1 Computing tree changes

⟨*signature* `Changes.changes_worktree_vs_index` 85f⟩≡ (150a)
```
(* for git diff and git status *)
val changes_worktree_vs_index:
  (Blob.hash -> Change.content) ->
  Fpath.t -> Index.t -> Change.t list
```

⟨*function* Changes.changes_worktree_vs_index 86a⟩≡ (150b)

```
(* less: could factorize with Diff_tree.changes_tree_vs_tree? would need
 * to generate flat list of files (but then less opti opportunity
 * in changes_tree_vs_tree when hash for a whole subtree is the same)
 * and then just do set differences to compute new, deleted, and
 * for changes just look intersection and check if same content.
 *)
let changes_worktree_vs_index read_blob worktree index =
  index |> List.map (fun entry ->
    let old_stat = entry.Index.stats in
    let path = worktree // entry.Index.path in
    let new_stat_opt =
      try Some (Unix.lstat !!path |> Index.stat_info_of_lstats)
      with Unix.Unix_error _ -> None
    in
    match new_stat_opt with
    | None ->
      [Change.Del { Change.path = entry.Index.path;
                    mode = old_stat.Index.mode;
                    content = lazy (read_blob entry.Index.id);
                  }]
    | Some new_stat ->
      (match () with
      (* useful opti? *)
      | _ when new_stat.Index.mtime = old_stat.Index.mtime -> []
      (* a change of mode is converted in a del/add *)
      | _ when new_stat.Index.mode <> old_stat.Index.mode ->
        [Change.Del { Change.path = entry.Index.path;
                      mode = old_stat.Index.mode;
                      content = lazy (read_blob entry.Index.id)};
         Change.Add { Change.path = entry.Index.path;
                      mode = new_stat.Index.mode;
                      content = lazy
                        (content_from_path_and_stat_index path new_stat)}
        ]
      | _ ->
        [Change.Modify (
          { Change.path = entry.Index.path;
            mode = old_stat.Index.mode;
            content = lazy (read_blob entry.Index.id) },
          { Change.path = entry.Index.path;
            mode = new_stat.Index.mode;
            content = lazy
              (content_from_path_and_stat_index path new_stat) }
        )]
      )
  ) |> List.flatten
```

Uses Change.entry.content 84e, Change.entry.mode 84e, Change.entry.path 84e, Change.t.Add 84d, Change.t.Del 84d, Change.t.Modify 84d, Changes.content_from_path_and_stat_index() 86b, Index.entry.id 39c, Index.entry.path 39c, Index.entry.stats 39c, Index.stat_info.mode 39d, and Index.stat_info_of_lstats() 173b.

⟨*function* Changes.content_from_path_and_stat_index 86b⟩≡ (150b)

```
(* similar to Repository.content_from_path_and_unix_stat *)
let content_from_path_and_stat_index (path : Fpath.t) stat_info =
  match stat_info.Index.mode with
  | Index.Link ->
    Unix.readlink !!path
  | Index.Normal | Index.Exec ->
      path |> UChan.with_open_in (fun (ch : Chan.i) ->
        ch.ic |> IO.input_channel |> IO.read_all
```

```
    )
```
⟨Changes.content_from_path_and_stat_index() *match mode cases* 133c⟩

Uses Common.with_file_in(), IO.input_channel(), IO.read_all(), Index.mode.Exec 39e, Index.mode.Link 39e, Index.mode.Normal 39e, and Index.stat_info.mode 39d.

12.3.2 Computing file changes

⟨*signature* Diffs.diff 87a⟩≡ (166a)
```
val diff: string -> string -> Diff.diff
```

12.3.3 Showing changes

⟨*signature* Diff_unified.show_change 87b⟩≡ (167a)
```
val show_change: Change.t -> unit
```

⟨*function* Diff_unified.show_change 87c⟩≡ (167b)
```
let show_change change =
  (* less: if mode is gitlink? *)
  let (old_path, old_content), (new_path, new_content) =
    match change with
    | Change.Add entry ->
      (Fpath.v "dev/null", lazy ""),
      (Fpath.v "b" // entry.Change.path, entry.Change.content)
    | Change.Del entry ->
      (Fpath.v "a" // entry.Change.path, entry.Change.content),
      (Fpath.v "dev/null", lazy "")
    | Change.Modify (entry1, entry2) ->
      (Fpath.v "a" // entry1.Change.path, entry1.Change.content),
      (Fpath.v "b" // entry2.Change.path, entry2.Change.content)
  in
  let diffs = Diffs.diff (Lazy.force old_content) (Lazy.force new_content) in
  if not (diffs |> List.for_all (function Diff.Equal _ -> true | _ -> false))
  then begin
    UConsole.print (spf "diff --git %s %s" !!old_path !!new_path);
    (* less: display change of modes *)
    show_unified_diff diffs
  end
```

Uses Change.entry.content 84e, Change.entry.path 84e, Change.t.Add 84d, Change.t.Del 84d, Change.t.Modify 84d, Common.pr(), Common.spf(), Diff.diff_elem.Equal, Diff_unified.show_unified_diff(), and Diffs.diff() 128c.

⟨*function* Diff_unified.show_unified_diff 87d⟩≡ (167b)
```
let show_unified_diff diffs =
  (* naive: no contextual:  diffs |> List.iter print *)
  let rec aux context_lines nctx_before nctx_after nold nnew diffs =
    match diffs with
    (* todo: say if 'No newline at end of file' *)
    | [] -> ()
    | x::xs ->
      (match x with
      | Diff.Equal _s ->
        (match () with
        | _ when nctx_after > 0 ->
          print x;
          aux [] 0 (nctx_after - 1) (nold + 1) (nnew + 1) xs
        | _ when nctx_before < nContext ->
          aux (x::context_lines) (nctx_before + 1) 0 (nold + 1) (nnew + 1) xs
        | _ when nctx_before = nContext ->
```

```
            let new_context_lines = List_.take nContext (x::context_lines) in
            aux new_context_lines nContext 0 (nold + 1) (nnew + 1) xs
          | _ -> raise (Impossible "")
          )
        | Diff.Deleted _s  ->
          let prevs = List_.take nctx_before context_lines |> List.rev in
          if prevs <> [] then print_header nctx_before nold nnew;
          prevs |> List.iter print;
          print x;
          aux [] 0 nContext (nold + 1) (nnew) xs
        | Diff.Added _s ->
          let prevs = List_.take nctx_before context_lines |> List.rev in
          if prevs <> [] then print_header nctx_before nold nnew;
          prevs |> List.iter print;
          print x;
          aux [] 0 nContext (nold) (nnew+1) xs
        )
    in
    aux [] 0 0 1 1 diffs
```

Uses `Common.List_.take()`, `Diff.diff_elem.Added` 85b, `Diff.diff_elem.Deleted` 85b, `Diff.diff_elem.Equal`, `Diff_unified.nContext`, `Diff_unified.print()`, and `Diff_unified.print_header()`.

⟨*constant* `Diff_unified.nContext` 88a⟩≡ (167b)

```
  let nContext = 3
```

⟨*function* `Diff_unified.print` 88b⟩≡ (167b)

```
  let print = function
    | Diff.Equal s ->
      print_string (" " ^ s)
    | Diff.Deleted s ->
      print_string ("-" ^ s)
    | Diff.Added s ->
      print_string ("+" ^ s)
```

Uses `Diff.diff_elem.Added` 85b, `Diff.diff_elem.Deleted` 85b, and `Diff.diff_elem.Equal`.

⟨*function* `Diff_unified.print_header` 88c⟩≡ (167b)

```
  let print_header nctx_before nold nnew =
    (* todo: should print size of hunk also here, but then
     * need to wait we finished processing this hunk
     *)
    print_string (spf "@@ -%d, +%d, @@\n"
                    (nold - nctx_before) (nnew - nctx_before))
```

Uses `Common.spf()`.

12.4 Commit history walker

12.4.1 Basic walker

⟨*signature* `Commit.walk_history` 88d⟩≡ (159a)

```
  val walk_history:
    (hash -> t) -> (hash -> t -> unit) -> hash ->
    unit
```

⟨*function* `Commit.walk_history` 89a⟩≡ (159b)
```
(* less: sort by time? so have a sorted queue of commits *)
let walk_history read_commit f sha =
  (* we are walking a DAG, so we need to remember already processed nodes *)
  let hdone = Hashtbl.create 101 in
  let rec aux sha =
    if Hashtbl.mem hdone sha
    then ()
    else begin
      Hashtbl.add hdone sha true;
      let commit = read_commit sha in
      (* todo: path matching *)
      f sha commit;
      commit.parents |> List.iter aux
    end
  in
  aux sha
(*
let walk_graph r f =
  let heads =
    Repository.all_refs r |> Common.map_filter (fun aref ->
      if aref =~ "refs/heads/"
      then Some (Repository.follow_ref_some r (Refs.Ref aref))
      else None
    )
  in
  ...
  heads |> List.iter aux
*)
```
Uses `Commit.t.parents` 36e.

12.4.2 Extra features

max entries

Reverse walking

Exclude set

Time range

Path restrictions

Topological order

12.5 Showing the commit history: `git log`

⟨*constant* `Cmd_log.cmd` 89b⟩≡ (155b)
```
let cmd = { Cmd_.
  name = "log";
  usage = " [options]";
  options = [
    "--name-status", Arg.Set name_status,
    " print name/status for each changed file";
    (* todo: -1, -10 *)
    (* less: --reverse *)
  ];
  f = (fun args ->
    let r, relpaths = Repository.find_root_open_and_adjust_paths (Fpath_.of_strings args) in
```

```
    match relpaths with
    | [] -> log r
    (* todo: git log path *)
    (* less: revision range *)
    | _xs -> raise Cmd_.ShowUsage
  );
}
```

Uses Cmd.t.f 55a, Cmd.t.options 55a, Cmd.t.usage 55a, Cmd_log.log() 90a, Cmd_log.name_status, and Repository.find_root_open_ 61e.

⟨*function* Cmd_log.log 90a⟩≡ (155b)

```
(* todo: track only selected paths
 * (and then rename detection to track correctly)
 *)
let log r =
  let start = Repository.follow_ref_some r (Refs.Head) in
  start |> Commit.walk_history (Repository.read_commit r) (fun sha commit ->
    print_commit sha commit;
    ⟨Cmd_log.log() if --name-status flag 90d⟩
    end
  )
```

Uses Cmd_log.print_commit(), Commit.walk_history() 89a, Refs.t.Head 37g, Repository.follow_ref_some() 46d, and Repository.read_commit() 45c.

12.5.1 Printing a commit

⟨*function* Cmd_log.print_commit 90b⟩≡ (155b)

```
let print_commit sha commit =
  UConsole.print (spf "commit: %s" (Hexsha.of_sha sha));
  (match commit.Commit.parents with
  | [] | [_] -> ()
  | _x::xs ->
    UConsole.print (spf "merge: %s"
          (xs |> List.map Hexsha.of_sha |> String.concat "..."));
  );
  let author = commit.Commit.author in
  UConsole.print (spf "Author: %s <%s>" author.User.name author.User.email);
  let committer = commit.Commit.committer in
  if author <> committer
  then
    UConsole.print (spf "Committer: %s <%s>" committer.User.name committer.User.email);
  UConsole.print (spf "Date:   %s" (User.string_of_date author.User.date));
  UConsole.print "";
  UConsole.print ("    " ^ commit.Commit.message);
  ()
```

Uses Commit.t.author 36e, Commit.t.committer 37c, Commit.t.message 36e, Commit.t.parents 36e, Common.pr(), Common.spf(), Hexsha.of_sha(), User.string_of_date() 84b, User.t.date 37a, User.t.email 37a, and User.t.name 37a.

12.5.2 Diff summary: `git log --name-status`

⟨*constant* Cmd_log.name_status 90c⟩≡ (155b)

```
let name_status = ref false
```

⟨Cmd_log.log() *if* --name-status *flag* 90d⟩≡ (90a)

```
if !name_status
then begin
  let tree1 = Repository.read_tree r commit.Commit.tree in
```

```
  let tree2 =
    match commit.Commit.parents with
    | [] -> []
    | [sha] ->
      let commit2 = Repository.read_commit r sha in
      Repository.read_tree r commit2.Commit.tree
    | _x::_y::_xs ->
      failwith "TODO: log: handle merge"
  in
  let changes = Changes.changes_tree_vs_tree
    (Repository.read_tree r)
    (Repository.read_blob r)
    tree2
    tree1
  in
  changes |> List.iter print_change;
  UConsole.print "";
```

Uses Changes.changes_tree_vs_tree() 91c, Cmd_log.print_change(), Commit.t.parents 36e, Commit.t.tree 36e, Common.pr(), Repository.read_blob() 42g, Repository.read_commit() 45c, and Repository.read_tree() 43f.

⟨*function* Cmd_log.print_change 91a⟩≡ (155b)

```
let print_change change =
  match change with
  | Change.Add entry ->
    UConsole.print (spf "A       %s" !!(entry.Change.path))
  | Change.Del entry ->
    UConsole.print (spf "D       %s" !!(entry.Change.path))
  | Change.Modify (entry1, _entry2) ->
    UConsole.print (spf "M       %s" !!(entry1.Change.path))
```

Uses Change.entry.path 84e, Change.t.Add 84d, Change.t.Del 84d, Change.t.Modify 84d, Common.pr(), and Common.spf().

12.5.3 Comparing trees

⟨*signature* Changes.changes_tree_vs_tree 91b⟩≡ (150a)

```
(* for git show commit *)
val changes_tree_vs_tree:
  (Tree.hash -> Tree.t) ->
  (Blob.hash -> Change.content) ->
  Tree.t -> Tree.t -> Change.t list
```

⟨*function* Changes.changes_tree_vs_tree 91c⟩≡ (150b)

```
(* see also Cmd_diff.changes_index_vs_worktree
 *     and  Cmd_status.changes_index_vs_HEAD
 *)
let changes_tree_vs_tree read_tree read_blob tree1 tree2 =
  let changes = ref [] in
  let add x = Stack_.push x changes in
  Tree.walk_trees read_tree (Fpath.v "XXX") (fun dirpath entry1_opt entry2_opt ->
    (* if entries are directories, then we would be called again
     * with their individual files, so safe to skip the dir entries.
     *)
    let entry1_opt = skip_tree_and_adjust_path read_blob dirpath entry1_opt in
    let entry2_opt = skip_tree_and_adjust_path read_blob dirpath entry2_opt in

    match entry1_opt, entry2_opt with
    | None, None -> ()
    | Some (a, asha), Some (b, bsha) ->
      (match () with
```

```
      (* file type changed reported as delete/add (meh) *)
      | _ when a.Change.mode <> b.Change.mode ->
        add (Change.Del a);
        add (Change.Add b);
      | _ when asha <> bsha ->
        add (Change.Modify (a, b))
      | _ -> ()
      )
    | Some (a,_), None -> add (Change.Del a)
    | None, Some (b,_) -> add (Change.Add b)
  ) tree1 tree2 ;
  List.rev !changes
```

Uses `Change.t.Add` 84d, `Change.t.Del` 84d, `Change.t.Modify` 84d, `Changes.skip_tree_and_adjust_path()`, `Common.push()`, and `Tree.walk_trees()` 183b.

⟨*signature* `Tree.walk_trees` 92a⟩≡ (183a)

```
val walk_trees:
  (hash -> t) -> Fpath.t (* dir *) ->
  (Fpath.t -> entry option -> entry option -> unit) -> t -> t -> unit
```

⟨*function* `Tree.walk_trees` 92b⟩≡ (183b)

```
let rec walk_trees read_tree (dirpath : Fpath.t) f xs ys =
  let g dirpath entry1_opt entry2_opt =
    f dirpath entry1_opt entry2_opt;
    (match entry1_opt, entry2_opt with
    | Some { perm = Dir; name = str; id = sha }, None ->
      walk_trees read_tree (dirpath / str) f
        (read_tree sha) []
    | None, Some { perm = Dir; name = str; id = sha } ->
      walk_trees read_tree (dirpath / str) f
        [] (read_tree sha)
    | Some { perm = Dir; name = str1; id = sha1 },
      Some { perm = Dir; name = str2; id = sha2 } ->
      assert (str1 = str2);
        (* todo: could skip if sha1 = sha2 here, useful opti *)
        walk_trees read_tree (dirpath / str1) f
          (read_tree sha1) (read_tree sha2)
    | None, None -> raise (Impossible "two None in walk_trees.g")
    (* no directories, no need to recurse *)
    | Some _, None
    | None, Some _
    | Some _, Some _
      -> ()
    )
  in
  match xs, ys with
  | [], [] -> ()
  | x::xs, [] ->
    g dirpath (Some x) None;
    walk_trees read_tree dirpath f xs ys
  | [], y::ys ->
    g dirpath None (Some y);
    walk_trees read_tree dirpath f xs ys
  | x::xs, y::ys ->
    (match x.name <=> y.name with
    | Equal ->
      g dirpath (Some x) (Some y);
      walk_trees read_tree dirpath f xs ys
    | Less ->
      g dirpath (Some x) None;
```

```
      walk_trees read_tree dirpath f xs (y::ys)
    | Greater ->
      g dirpath None (Some y);
      walk_trees read_tree dirpath f (x::xs) ys
    )
```

Uses Common.<=>(), Common.compare.Equal, Common.compare.Inf, Common.compare.Sup, Tree.entry.id 36b, Tree.entry.name 36b, Tree.entry.perm 36b, Tree.perm.Dir 36d, and Tree.walk_trees() 183b.

⟨*function* Changes.skip_tree_and_adjust_path 93a⟩≡ (150b)

```
let skip_tree_and_adjust_path read_blob dirpath entry_opt =
  match entry_opt with
  | Some { Tree.perm = Tree.Dir; _ } -> None
  | Some { Tree.perm = Tree.Commit; _ } -> failwith "submodule not supported"
  | Some x -> Some ({ Change.
    path = dirpath / x.Tree.name;
    mode = Index.mode_of_perm x.Tree.perm;

    (* todo: do that later? once know we will return a change with this entry?
     * make it lazy?
     *)
    content = lazy (read_blob x.Tree.id);
  }, x.Tree.id)
  | None -> None
```

Uses Change.entry.content 84e, Change.entry.mode 84e, Change.entry.path 84e, Index.mode_of_perm() 173b, Tree.entry.id 36b, Tree.entry.name 36b, Tree.entry.perm 36b, Tree.perm.Commit 132a, and Tree.perm.Dir 36d.

⟨*signature* Index.mode_of_perm 93b⟩≡ (173a)

```
val mode_of_perm: Tree.perm -> mode
```

⟨*function* Index.mode_of_perm 93c⟩≡ (173b)

```
let mode_of_perm perm =
  match perm with
  | Tree.Normal -> Normal
  | Tree.Exec -> Exec
  | Tree.Link -> Link
  ⟨Index.mode_of_perm() match perm cases 133e⟩
  | Tree.Dir -> failwith "index entry does not support Tree.dir perm"
```

Uses Index.mode.Exec 39e, Index.mode.Gitlink 132b, Index.mode.Link 39e, Index.mode.Normal 39e, Tree.perm.Commit 132a, Tree.perm.Exec 36d, Tree.perm.Link 36d, and Tree.perm.Normal 36d.

12.6 Showing the status of a file: `git status`

⟨*constant* Cmd_status.cmd 93d⟩≡ (156d)

```
let cmd = { Cmd_.
  name = "status";
  usage = " [options]"; (* less: <pathspec> *)
  options = [
    "--short", Arg.Set short_format, " show status concisely";
    "--long", Arg.Clear short_format, " show status in long format (default)";
    (* less: --branch, --ignored *)
  ];
  f = (fun args ->
    let r, relpaths = Repository.find_root_open_and_adjust_paths (Fpath_.of_strings args) in
    match relpaths with
    | [] -> status r
    | _xs -> raise Cmd_.ShowUsage
  );
}
```

Uses Cmd.t.f 55a, Cmd.t.options 55a, Cmd.t.usage 55a, Cmd_status.short_format 156d, Cmd_status.status() 156d 156d, and Repository.find_root_open_and_adjust_paths() 61e.

⟨*function* Cmd_status.status 94a⟩≡ (156d)
```
let status r =
  let st = status_of_repository r in
  if !short_format
  then print_status_short st
  else print_status_long st
```
Uses Cmd_status.print_status_long() 156d, Cmd_status.print_status_short() 156d, and Cmd_status.short_format 156d.

⟨*constant* Cmd_status.short_format 94b⟩≡ (156d)
```
let short_format = ref false
```

⟨*function* Cmd_status.print_status_short 94c⟩≡ (156d)
```
let print_status_short _st =
  raise Todo
```

⟨*function* Cmd_status.print_status_long 94d⟩≡ (156d)
```
let print_status_long st =
  if st.staged <> []
  then begin
    UConsole.print "Changes to be committed:";
(*  (use "git reset HEAD <file>..." to unstage) *)
    UConsole.print "";
    st.staged |> List.iter print_change_long;
    UConsole.print "";
  end;
  if st.unstaged <> []
  then begin
    UConsole.print "Changes not staged for commit:";
    UConsole.print "";
(*
  (use "git add/rm <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)
*)

    st.unstaged |> List.iter print_change_long;
    UConsole.print "";
  end;
  if st.untracked <> []
  then begin
    UConsole.print "Untracked files:";
(*  (use "git add <file>..." to include in what will be committed) *)
    UConsole.print "";
    st.untracked |> List.iter (fun file ->
      UConsole.print (spf " %s" !!file)
    );
    UConsole.print "";
  end
```
Uses Cmd_status.print_change_long(), Cmd_status.status.staged 94e, Cmd_status.status.unstaged 94e, Cmd_status.status.untracked 94e, Common.pr(), and Common.spf().

⟨*type* Cmd_status.status 94e⟩≡ (156d)
```
type status = {
  (* diff index vs HEAD *)
  staged: Change.t list;
  (* diff worktree vs index *)
  unstaged: Change.t list;
  (* other *)
```

```
    untracked: Fpath.t list;
  }
```

Uses Change.t 84d.

⟨*function* Cmd_status.print_change_long 95a⟩≡ (156d)

```
  (* very similar to Cmd_log.print_change, but with more indentation *)
  let print_change_long change =
    match change with
    | Change.Add entry ->
      UConsole.print (spf " new file: %s" !!(entry.Change.path))
    | Change.Del entry ->
      UConsole.print (spf " deleted: %s" !!(entry.Change.path))
    | Change.Modify (entry1, _entry2) ->
      UConsole.print (spf " modified: %s" !!(entry1.Change.path))
```

Uses Change.entry.path 84e, Change.t.Add 84d, Change.t.Del 84d, Change.t.Modify 84d, Common.pr(), and Common.spf().

⟨*function* Cmd_status.status_of_repository 95b⟩≡ (156d)

```
  let status_of_repository r =
    { staged = changes_index_vs_HEAD r;
      unstaged =
        Changes.changes_worktree_vs_index
          (Repository.read_blob r)
          r.Repository.worktree
          r.Repository.index;
      untracked = untracked r;
    }
```

Uses Changes.changes_worktree_vs_index() 86a, Cmd_status.status.unstaged 94e, Cmd_status.status.untracked 94e, Cmd_status.untracked(), Repository.read_blob() 42g, Repository.t.index, and Repository.t.worktree 35a.

12.6.1 Listing staged modifications

⟨*function* Cmd_status.changes_index_vs_HEAD 95c⟩≡ (156d)

```
  let changes_index_vs_HEAD r =
    let commitid = Repository.follow_ref_some r (Refs.Head) in
    let commit = Repository.read_commit r commitid in
    let treeid = commit.Commit.tree in
    Changes.changes_index_vs_tree (Repository.read_tree r)
      r.Repository.index
      treeid
```

Uses Changes.changes_index_vs_tree() 95e, Commit.t.tree 36e, Repository.read_commit() 45c, Repository.read_tree() 43f, and Repository.t.index.

⟨*signature* Changes.changes_index_vs_tree 95d⟩≡ (150a)

```
  (* for git status (to compare index vs HEAD) *)
  val changes_index_vs_tree:
    (Tree.hash -> Tree.t) ->
    Index.t -> Tree.hash -> Change.t list
```

⟨*function* Changes.changes_index_vs_tree 95e⟩≡ (150b)

```
  (* some commonalities with Repository.set_worktree_and_index_to_tree *)
  let changes_index_vs_tree read_tree index treeid =
    let tree = read_tree treeid in

    let h_in_index_and_head = Hashtbl.create 101 in
    let hindex =
      index
      |> List.map (fun entry -> entry.Index.path, entry)
      |> Hashtbl_.of_list
```

```
  in
  let changes = ref [] in

  tree |> Tree.walk_tree read_tree (Fpath.v "XXX") (fun relpath entry_head ->
    let perm = entry_head.Tree.perm in
    match perm with
    | Tree.Dir -> ()
    | Tree.Commit -> failwith "submodule not yet supported"
    | Tree.Normal | Tree.Exec | Tree.Link ->
      try
        let entry_index = Hashtbl.find hindex relpath in
        Hashtbl.add h_in_index_and_head relpath true;
        (* less: if change mode, then report as del/add *)
        if entry_head.Tree.id <> entry_index.Index.id
        then changes |> Stack_.push (Change.Modify (
          { Change.path = relpath;
            mode = Index.mode_of_perm perm;
            content = lazy (raise (Impossible "not called")); },
          { Change.path = relpath;
            mode = entry_index.Index.stats.Index.mode;
            content = lazy (raise (Impossible "not called")); }
        ))
      with Not_found ->
        changes |> Stack_.push (Change.Del { Change.
             path = relpath;
             mode = Index.mode_of_perm perm;
             content = lazy (raise (Impossible "not called"));
                                          });
  );
  index |> List.iter (fun entry ->
    if not (Hashtbl.mem h_in_index_and_head entry.Index.path)
    then changes |> Stack_.push (Change.Add { Change.
             path = entry.Index.path;
             mode = entry.Index.stats.Index.mode;                                    content = lazy (ra
    )
  );
  (* less: sort by path *)
  List.rev !changes
```

Uses `Change.entry.content` 84e, `Change.entry.mode` 84e, `Change.entry.path` 84e, `Change.t.Add` 84d, `Change.t.Del` 84d, `Change.t.Modify` 84d, `Common.Hashtbl_.of_list()`, `Common.push()`, `Index.entry.id` 39c, `Index.entry.path` 39c, `Index.entry.stats` 39c, `Index.mode_of_perm()` 173b, `Index.stat_info.mode` 39d, `Tree.entry.id` 36b, `Tree.entry.perm` 36b, `Tree.perm.Commit` 132a, `Tree.perm.Dir` 36d, `Tree.perm.Exec` 36d, `Tree.perm.Link` 36d, `Tree.perm.Normal` 36d, and `Tree.walk_tree()` 78b.

12.6.2 Listing unstaged modifications

12.6.3 Listing untracked files

⟨*function* `Cmd_status.untracked` 96⟩≡ (156d)

```
  (* todo: need parse .gitignore *)
  let untracked r =
    let h = r.Repository.index
        |> List.map (fun entry -> entry.Index.path, true)
        |> Hashtbl_.of_list
    in
    let res = ref [] in
    r.Repository.worktree |> Repository.walk_dir (fun dir _dirs files ->
      files |> List.iter (fun file ->
```

```
      let path = dir / file in
      let path =
        if !!path =~ "^\\./\\(.*\\)"
        then Fpath.v (Regexp_.matched1 !!path)
        else path
      in
      if not (Hashtbl.mem h path)
      then Stack_.push path res
    );
  );
  List.rev !res
```

Uses Common.Hashtbl_.of_list(), Common.Regexp_.matched1(), Common.push(), Index.entry.path 39c, Repository.t.index, Repository.t.worktree 35a, and Repository.walk_dir() 145b.

12.7 Archeology

12.7.1 Checking out an old version

⟨Cmd_checkout.checkout() *cases* 97⟩≡ (76e)

```
  | _ when Hexsha.is_hexsha str ->
    let commitid = Hexsha.to_sha str in
    let commit = Repository.read_commit r commitid in
    let treeid = commit.Commit.tree in
    let tree = Repository.read_tree r treeid in
    (* todo: order of operation? set ref before index? reverse? *)
    Repository.write_ref r (Refs.Head) (Refs.Hash commitid);
    Repository.set_worktree_and_index_to_tree r tree;
    UConsole.print (spf "Note: checking out '%s'." str);
    UConsole.print ("You are in 'detached HEAD' state");
```

Uses Commit.t.tree 36e, Common.pr(), Common.spf(), Hexsha.to_sha(), Refs.ref_content.Hash 38b, Refs.t.Head 37g, Repository.read_commit() 45c, Repository.read_tree() 43f, Repository.set_worktree_and_index_to_tree() 77d, and Repository.write_ref() 52g.

12.7.2 Showing an old version of a file

12.7.3 Showing differences between past versions

12.7.4 Showing the history of a file or directory: `git log <path>`

Chapter 13

Packing

13.1 `Pack.t`

13.1.1 Pack data

13.1.2 Pack Index

Index pack v1

Index pack v2

13.2 IO

13.2.1 Reading a pack

13.2.2 Writing a pack

13.3 Modifications to previous functions and data structures

13.3.1 Object Store

13.3.2 Refs

13.4 Delta

13.5 Pack commands

13.5.1 `git pack-objects`

13.5.2 `git repack`

13.5.3 `git fetch-pack`

13.5.4 `git receive-pack`

Chapter 14

Exchanging

14.1 Client/server architecture

⟨*type* `Client.t` 99a⟩≡ (151)
```
type t = {
  (* path to remote (e.g., /path/other/repo, or git://github.com/foo/bar)
   * Fpath.t or Uri.t (TODO: use variant and enforce?)
  *)
  url: string;
  (* less: more parameters:
   *  - determine_refs_wanted. for now grabs everything from remote HEAD
   *  - return set of remote refs, not just the one for HEAD
   * Note that fetch will modify the target repository by side effect.
   *)
  fetch: Repository.t -> Commit.hash;
  (* less: progress *)
}
```
Uses `Repository.t`.

⟨*signature* `Clients.client_of_url` 99b⟩≡ (152b)
```
val client_of_url: string -> Client.t
```

⟨*function* `Clients.client_of_url` 99c⟩≡ (152c)
```
(* old: was called get_transport_and_path (and xxx_from_url) in dulwich *)
let client_of_url url =
  match url with
  ⟨Clients.client_of_url() match url cases 108a⟩
  | s ->
    if Sys.file_exists s
    then Client_local.mk_client (Fpath.v s)
    else failwith (spf "remote repository URL not supported: %s" url)
```
Uses `Client_local.mk_client()` 152a and `Common.spf()`.

14.2 Pulling updates from another repository: `git pull`

⟨*constant* `Cmd_pull.cmd` 99d⟩≡ (155d)
```
let cmd = { Cmd_.
  name = "pull";
  usage = " [options] [<url repository>]";
  options = [
  ];
  f = (fun args ->
    let dst, _ = Repository.find_root_open_and_adjust_paths [] in
```

```
      match args with
      | [] ->
        failwith "TODO: use remote information in config file"
      | [url] ->
        pull dst url
      | _ -> raise Cmd_.ShowUsage
    );
  }
```

Uses Cmd.t.f 55a, Cmd.t.name 55a, Cmd.t.options 55a, Cmd.t.usage 55a, Cmd_pull.pull() 100a, and Repository.find_root_open_and 61e.

⟨*function* Cmd_pull.pull 100a⟩≡ (155d)

```
  (* =~ git fetch + git merge *)
  let pull dst url =
    (* todo: detect if clean repo? status is empty? *)
    let client = Clients.client_of_url url in

    (* less: allow to grab from multiple heads, not just HEAD *)
    let remote_HEAD_sha = client.Client.fetch dst in

    (* detect if need merge, if current HEAD not parent of new HEAD *)
    let current_HEAD_sha = Repository.follow_ref_some dst (Refs.Head) in
    let ancestors_remote_HEAD =
      Commit.collect_ancestors (Repository.read_commit dst) [remote_HEAD_sha]
        (Hashtbl.create 101)
    in
    (match () with
    | _ when remote_HEAD_sha = current_HEAD_sha -> ()
    | _ when Hashtbl.mem ancestors_remote_HEAD current_HEAD_sha ->
      (* easy case *)
      UConsole.print (spf "fast forward to %s" (Hexsha.of_sha remote_HEAD_sha));
      Repository.set_ref dst (Refs.Head) remote_HEAD_sha;
      let commit = Repository.read_commit dst remote_HEAD_sha in
      let tree = Repository.read_tree dst (commit.Commit.tree) in
      Repository.set_worktree_and_index_to_tree dst tree
    | _ -> failwith "TODO: git pull need merge"
    )
```

Uses Client.t.fetch 99a, Clients.client_of_url() 99c, Commit.collect_ancestors() 103c, Commit.t.tree 36e, Common.pr(), Common.spf(), Hexsha.of_sha(), Refs.t.Head 37g, Repository.follow_ref_some() 46d, Repository.read_commit() 45c, Repository.read_tree() 43f, Repository.set_ref() 100c, and Repository.set_worktree_and_index_to_tree() 77d.

⟨*signature* Repository.set_ref 100b⟩≡ (179)

```
  (* better than write_ref, will follow symbolic ref *)
  val set_ref: t -> Refs.t -> Commit.hash -> unit
```

⟨*function* Repository.set_ref 100c⟩≡ (180)

```
  let set_ref r aref newh =
    let (refs, _) = follow_ref r aref in
    let lastref = List.hd (List.rev refs) in
    let file = ref_to_filename r lastref in
    file |> with_file_out_with_lock (fun ch ->
      ch |> IO.output_channel |> IO_.with_close_out
          (Refs.write (Refs.Hash newh))
    )
```

Uses IO.output_channel(), IO_.with_close_out() 146c, Refs.ref_content.Hash 38b, Refs.write(), Repository.follow_ref() 46b, Repository.ref_to_filename() 38e, and Repository.with_file_out_with_lock() 50d.

⟨*signature* Client_local.mk_client 100d⟩≡ (151e)

```
  val mk_client: Fpath.t -> Client.t
```

⟨*function* `Client_local.mk_client` 101a⟩≡ (152a)
```
let mk_client (path : Fpath.t) =
  { Client.
    url = !!path;
    fetch = (fun dst ->
      let src = Repository.open_ path in
      fetch_objects src dst;
      (* less: all_refs *)
      Repository.follow_ref_some src Refs.Head
   );
  }
```
Uses `Client.t.fetch` 99a, `Client_local.fetch_objects()` 152a, `Refs.t.Head` 37g, `Repository.follow_ref_some()` 46d, and `Repository.open_()` 61b.

14.2.1 Fetching objects locally

⟨*function* `Client_local.fetch_objects` 101b⟩≡ (152a)
```
let fetch_objects src dst =
  (* less: determine_wants from pull command *)
  let top_wanted_commits = [Repository.follow_ref_some src Refs.Head] in
  (* less: shallows? unshallows? *)
  let top_common_commits = find_top_common_commits src dst in
  iter_missing_objects top_common_commits top_wanted_commits src
    (fun sha1 obj_opt ->
    (* less: opti: copy raw files directly without unmarshalling/marshalling *)
    let obj =
      match obj_opt with
      | None -> Repository.read_obj src sha1
      | Some obj -> obj
    in
    (* todo: count objects progress *)
    (* pr2 (spf "adding %s" (Hexsha.of_sha sha1)); *)
    let sha2 = Repository.add_obj dst obj in
    assert (sha1 = sha2)
  )
```
Uses `Client_local.find_top_common_commits()`, `Client_local.iter_missing_objects()` 152a, `Refs.t.Head` 37g, `Repository.add_obj()` 49b, `Repository.follow_ref_some()` 46d, and `Repository.read_obj()` 41b.

Find top common commits

⟨*function* `Client_local.find_top_common_commits` 101c⟩≡ (152a)
```
(* find the common frontline *)
let find_top_common_commits src dst =
  let top_commons = Hashtbl.create 101 in
  let walker = mk_graph_walker dst in

  let rec loop_while_sha commit_sha_opt =
    commit_sha_opt |> Option.iter (fun commit_sha ->
      if Repository.has_obj src commit_sha
      then begin
        Hashtbl.add top_commons commit_sha true;
        walker.ack commit_sha;
      end;
      loop_while_sha (walker.next ())
    )
  in
  loop_while_sha (walker.next ());
  top_commons |> Hashtbl_.to_list |> List.map fst
```

Uses Client_local.graph_walker.ack 104a, Client_local.graph_walker.next 104a, Client_local.mk_graph_walker() 152a, Common.Hashtbl_.to_list(), Common.if_some(), and Repository.has_obj() 102b.

⟨*signature* Repository.has_obj 102a⟩≡ (179)
```
val has_obj: t -> Sha1.t -> bool
```

⟨*function* Repository.has_obj 102b⟩≡ (180)
```
let has_obj r h =
  let path = h |> Hexsha.of_sha |> hexsha_to_filename r in
  Sys.file_exists !!path
```
Uses Hexsha.of_sha() and Repository.hexsha_to_filename() 35b.

Find missing objects

⟨*function* Client_local.iter_missing_objects 102c⟩≡ (152a)
```
let iter_missing_objects top_common_commits top_wanted_commits src f =
  (* less: split_commits_and_tags? *)
  let all_common_commits =
    Commit.collect_ancestors (Repository.read_commit src) top_common_commits
      (Hashtbl.create 101) in
  (* bugfix: do not forget Hashtbl.copy because collect_ancestors modify
   * the second parameter by side effect
   *)
  let missing_commits =
    Commit.collect_ancestors (Repository.read_commit src) top_wanted_commits
      (Hashtbl.copy all_common_commits)
  in

  (* let's iterate over all common commits *)

  let dst_have_sha = Hashtbl.create 101 in
  (* less: start from second returned result from collect_ancestors?
   * common_commits different from all_ancestors in VCS.nw?
   *)
  (* expensive loop below? so use parallel threads? *)
  all_common_commits |> Hashtbl.iter (fun commit_sha _true ->
    Hashtbl.add dst_have_sha commit_sha true;
    let commit = Repository.read_commit src commit_sha in
    collect_filetree (Repository.read_tree src) commit.Commit.tree dst_have_sha
  );

  (* and now let's iterate over all missing commits *)

  (* less: tags handling *)
  let rec missing sha is_blob =
    if Hashtbl.mem dst_have_sha sha
    then ()
    else begin
      Hashtbl.add dst_have_sha sha true;
      (if is_blob
       then f sha None
       else begin
        let obj = Repository.read_obj src sha in
        f sha (Some obj);
        (match obj with
        | Objects.Commit commit ->
          missing commit.Commit.tree false
        | Objects.Tree tree ->
          tree |> List.iter (fun entry ->
```

```
            if entry.Tree.perm = Tree.Commit
            then failwith "submodule not supported";
            (* bugfix: it's <>, not = *)
            missing entry.Tree.id (entry.Tree.perm <> Tree.Dir)
          )
        | Objects.Blob _ ->
          raise (Impossible "is_blob guard")
        )
       end
      );
  end
in
missing_commits |> Hashtbl.iter (fun commit_sha _true ->
  missing commit_sha false
)
```

Uses Client_local.collect_filetree() 152a, Commit.collect_ancestors() 103c, Commit.t.tree 36e, Objects.t.Blob 35d, Objects.t.Commit 35d, Objects.t.Tree 35d, Repository.read_commit() 45c, Repository.read_obj() 41b, Repository.read_tree() 43f, Tree.entry.id 36b, Tree.entry.perm 36b, Tree.perm.Commit 132a, and Tree.perm.Dir 36d.

⟨*function* Client_local.collect_filetree 103a⟩≡ (152a)

```
let rec collect_filetree read_tree treeid have_sha =
  let tree = read_tree treeid in
  tree |> List.iter (fun entry ->
    let sha = entry.Tree.id in
    if not (Hashtbl.mem have_sha sha) then begin
      Hashtbl.add have_sha sha true;
      match entry.Tree.perm with
      | Tree.Normal | Tree.Exec | Tree.Link -> ()
      | Tree.Dir ->  collect_filetree read_tree sha have_sha
      | Tree.Commit -> failwith "submodule not supported yet"
    end
  )
```

Uses Client_local.collect_filetree() 152a, Tree.entry.id 36b, Tree.entry.perm 36b, Tree.perm.Commit 132a, Tree.perm.Dir 36d, Tree.perm.Exec 36d, Tree.perm.Link 36d, and Tree.perm.Normal 36d.

⟨*signature* Commit.collect_ancestors 103b⟩≡ (159a)

```
val collect_ancestors:
  (hash -> t) ->  hash list -> hash Hashtbl_.set ->
  hash Hashtbl_.set
```

⟨*function* Commit.collect_ancestors 103c⟩≡ (159b)

```
(* similar to walk_history but with exposed hdone hash *)
let collect_ancestors read_commit top_commits hdone =
  let hcommits = Hashtbl.create 101 in
  let rec aux sha =
    if Hashtbl.mem hdone sha
    then ()
    else begin
      Hashtbl.add hdone sha true;
      Hashtbl.add hcommits sha true;
      let commit = read_commit sha in
      commit.parents |> List.iter aux
    end
  in
  top_commits |> List.iter aux;
  hcommits
```

Uses Commit.t.parents 36e.

Iterating over objects

Adding Objects

14.2.2 Fetching references

Refspecs

Selecting the refs

Parsing ref tuples

14.2.3 Fetching objects in a pack

14.3 Object store walker

⟨*type* `Client_local.graph_walker` 104a⟩≡ (152a)
```
(* will start from the heads and iterate over the ancestry of heads
 * until the caller ack that some top commits are already known and
 * do not need to be iterated furthermore.
 *)
type graph_walker = {
  next: unit -> Commit.hash option;
  ack: Commit.hash -> unit;
}
```

⟨*function* `Client_local.ml_graph_walker` 104b⟩≡ (152a)
```
let (mk_graph_walker: Repository.t -> graph_walker) = fun r ->
  (* less: start just from HEAD? *)
  let heads =
    Repository.all_refs r |> List.filter_map (fun aref ->
      if aref =~ "refs/heads/"
      then Some (Repository.follow_ref_some r (Refs.Ref aref))
      else None
    )
  in
  let todos = ref heads in
  let todos_next_round = ref [] in
  let last_round = ref None in
  let hdone = Hashtbl.create 101 in

  { next = (fun () ->
    todos := !todos_next_round @ !todos;
    todos_next_round := [];
    match !todos with
    | [] -> None
    | x::xs ->
      todos := xs;
      Hashtbl.add hdone x true;
      last_round := Some x;
      let commit = Repository.read_commit r x in
      let parents = commit.Commit.parents in
      parents |> List.iter (fun parent ->
        if Hashtbl.mem hdone parent
        then ()
        else todos_next_round := parent::!todos_next_round;
      );
      Some x
    );
```

```
    ack = (fun commit_sha ->
      (* less: do weird loop where recurse also over parents as in dulwich? *)
      match !last_round with
      | None -> raise (Impossible "ack always after at least one next");
      | Some x ->
        if x <> commit_sha
        then raise (Impossible "'ack(x)' should follow 'x = next()'");
        (* skip those one then because parent already in common *)
        todos_next_round := []
    );
  }
```

Uses Client_local.graph_walker.ack 104a, Client_local.graph_walker.next 104a, Commit.t.parents 36e, Common.map_filter(), Refs.t.Ref 37g, Repository.all_refs() 74b, Repository.follow_ref_some() 46d, and Repository.read_commit() 45c.

14.4 Pushing changes to another repository: `git push`

⟨*constant* Cmd_push.cmd 105a⟩≡ (155e)

```
let cmd = { Cmd_.
  name = "push";
  usage = " [options] [<url repository>]";
  options = [
    (* less: --all, --force, --progress *)
  ];
  f = (fun args ->
    let src, _ = Repository.find_root_open_and_adjust_paths [] in
    match args with
    | [] ->
      failwith "TODO: use remote information in config file"
    | [url] ->
      push src url
    | _ -> raise Cmd_.ShowUsage
  );
}
```

Uses Cmd.t.f 55a, Cmd.t.options 55a, Cmd.t.usage 55a, Cmd_push.push() 105b, and Repository.find_root_open_and_adjust_paths(61e.

⟨*function* Cmd_push.push 105b⟩≡ (155e)

```
(* =~ git fetch + git merge but inverting dst and src  *)
let push src_repo (url_dst : string) =
  let url = src_repo.Repository.worktree in
  let dst = Repository.open_ (Fpath.v url_dst) in
  (* todo: detect if clean repo? status is empty? *)
  let client = Clients.client_of_url !!url in


  (* less: allow to grab from multiple heads, not just HEAD *)
  let remote_HEAD_sha = client.Client.fetch dst in

  (* detect if need merge, if current HEAD not parent of new HEAD *)
  let current_HEAD_sha = Repository.follow_ref_some dst (Refs.Head) in
  let ancestors_remote_HEAD =
    Commit.collect_ancestors (Repository.read_commit dst) [remote_HEAD_sha]
      (Hashtbl.create 101)
  in
  (match () with
  | _ when current_HEAD_sha = remote_HEAD_sha -> ()
  | _ when Hashtbl.mem ancestors_remote_HEAD current_HEAD_sha ->
```

```
    (* easy case *)
    UConsole.print (spf "fast forward to %s" (Hexsha.of_sha remote_HEAD_sha));
    Repository.set_ref dst (Refs.Head) remote_HEAD_sha;
    let commit = Repository.read_commit dst remote_HEAD_sha in
    let tree = Repository.read_tree dst (commit.Commit.tree) in
    Repository.set_worktree_and_index_to_tree dst tree
  | _ -> failwith "TODO: git pull need merge"
  )
```

Uses Client.t.fetch 99a, Clients.client_of_url() 99c, Commit.collect_ancestors() 103c, Commit.t.tree 36e, Common.pr(), Common.spf(), Hexsha.of_sha(), Refs.t.Head 37g, Repository.follow_ref_some() 46d, Repository.open_() 61b, Repository.read_commit() 45c, Repository.read_tree() 43f, Repository.set_ref() 100c, Repository.set_worktree_and_index_to_tree() 77d, and Repository.t.worktree 35a.

14.4.1 Sending objects locally

14.4.2 Sending objects in a pack

14.5 Cloning a repository: `git clone`

⟨*constant* Cmd_clone.cmd 106a⟩≡ (154a)

```
let cmd = { Cmd_.
  name = "clone";
  usage = " [options] <repo> [<dir>]";
  options = [
    (* less: --bare, --progress, --depth *)
  ];
  f = (fun args ->
    match args with
    | [url]     -> clone url (Fpath.v ".")
    | [url;dst] -> clone url (Fpath.v dst)
    | _ -> raise Cmd_.ShowUsage
  );
}
```

Uses Cmd.t.f 55a, Cmd.t.name 55a, Cmd.t.options 55a, Cmd.t.usage 55a, and Cmd_clone.clone() 106b.

⟨*function* Cmd_clone.clone 106b⟩≡ (154a)

```
(* =~ git pull from scratch (itself =~ git fetch + git merge) *)
let clone url (path_dst : Fpath.t) =
  let client = Clients.client_of_url url in

  Repository.init path_dst;
  let dst = Repository.open_ path_dst in

  (* less: allow to grab from different head? *)
  let remote_HEAD_sha = client.Client.fetch dst in
  Repository.set_ref dst (Refs.Head) remote_HEAD_sha;

  (* =~ reset index *)
  let commit = Repository.read_commit dst remote_HEAD_sha in
  let tree = Repository.read_tree dst (commit.Commit.tree) in
  Repository.set_worktree_and_index_to_tree dst tree
```

Uses Client.t.fetch 99a, Clients.client_of_url() 99c, Commit.t.tree 36e, Refs.t.Head 37g, Repository.init() 59c, Repository.open_() 61b, Repository.read_commit() 45c, Repository.read_tree() 43f, and Repository.set_ref() 100c.

14.5.1 Fetching objects

14.5.2 Importing references

Chapter 15

Networking

⟨`Clients.client_of_url()` *match url cases* 108a⟩≡ (99c) 137a▷
```
  (* less: should use URL parsing library *)
  | s when s =~ "^git://" ->
    Client_git.mk_client url
```
Uses `Client_git.mk_client()` 108c.

15.1 `git://` protocol

15.1.1 Reading packets

15.1.2 Writing packets

15.2 Capabilities

15.3 `git://` client

⟨*signature* `Client_git.mk_client` 108b⟩≡ (151c)
```
  val mk_client: string (* Fpath.t or Uri.t like git:// *) -> Client.t
```

⟨*function* `Client_git.mk_client` 108c⟩≡ (151d)
```
  let mk_client _url =
    raise Todo
```

15.3.1 Fetching remote objects

15.3.2 Fetching pack files

`read_pkt_refs()`

`_handle_upload_pack_head()`

`_handle_upload_pack_tail()`

15.3.3 Sending pack files

15.4 git:// server

15.4.1 git-upload-pack

15.4.2 git-receive-pack

Chapter 16

Algorithms

16.1 SHA1

16.1.1 Overview

16.1.2 32 bits operators

⟨Sha1.sha1() *operator shortcuts for Int32* 110a⟩≡ (110b)

```
let ( lor )  = Int32.logor in
let ( lxor ) = Int32.logxor in
let ( land ) = Int32.logand in
let ( ++ )   = Int32.add in
let lnot     = Int32.lognot in
let sr       = Int32.shift_right in
let sl       = Int32.shift_left in
let cls n x = (sl x n) lor (Int32.shift_right_logical x (32 - n)) in
```

16.1.3 Entry point: sha1()

⟨*function* Sha1.sha1 110b⟩≡ (182b)

```
(* sha-1 digest. Based on pseudo-code of RFC 3174.
   Slow and ugly but does the job. *)
let sha1 s =
  ⟨function sha_1_pad 112a⟩
  let ( &&& ) = ( land ) in
  (* Operations on int32 *)
  ⟨Sha1.sha1() operator shortcuts for Int32 110a⟩
  (* Start *)
  let m = sha_1_pad s in
  let w = Array.make 16 0l in

  (* components of the 20 bytes SHA1 (5 * 4 bytes (Int32 with 'l' suffix)) *)
  let h0 = ref 0x67452301l in
  let h1 = ref 0xEFCDAB89l in
  let h2 = ref 0x98BADCFEl in
  let h3 = ref 0x10325476l in
  let h4 = ref 0xC3D2E1F0l in

  let a = ref 0l in
  let b = ref 0l in
  let c = ref 0l in
  let d = ref 0l in
  let e = ref 0l in
```

```
  (* For each block *)
  for i = 0 to ((Bytes.length m) / 64) - 1 do
    (* Fill w *)
    ⟨Sha1.sha1() fill w for each block i 111b⟩
    (* Loop *)
    a := !h0; b := !h1; c := !h2; d := !h3; e := !h4;
    ⟨Sha1.sha1() loop from 0 to 79 for each block i 111c⟩
    (* Update *)
    h0 := !h0 ++ !a;
    h1 := !h1 ++ !b;
    h2 := !h2 ++ !c;
    h3 := !h3 ++ !d;
    h4 := !h4 ++ !e
  done;

  (* the result hash number of 20 bytes *)
  let h = Bytes.create 20 in
  ⟨function i2s 111a⟩
  i2s h 0 !h0;
  i2s h 4 !h1;
  i2s h 8 !h2;
  i2s h 12 !h3;
  i2s h 16 !h4;
  Bytes.unsafe_to_string h
```

16.1.4 Endianess

⟨function i2s 111a⟩≡ (110b)

```
let i2s h k i =
  Bytes.set h k       (Char.unsafe_chr ((Int32.to_int (sr i 24)) &&& 0xFF));
  Bytes.set h (k + 1) (Char.unsafe_chr ((Int32.to_int (sr i 16)) &&& 0xFF));
  Bytes.set h (k + 2) (Char.unsafe_chr ((Int32.to_int (sr i 8))  &&& 0xFF));
  Bytes.set h (k + 3) (Char.unsafe_chr ((Int32.to_int i)         &&& 0xFF));
in
```

16.1.5 Filling blocks

⟨Sha1.sha1() fill w for each block i 111b⟩≡ (110b)

```
let base = i * 64 in
for j = 0 to 15 do
  let k = base + (j * 4) in
  w.(j) <- sl (Int32.of_int (Char.code @@ Bytes.get m k)) 24 lor
           sl (Int32.of_int (Char.code @@ Bytes.get m (k + 1))) 16 lor
           sl (Int32.of_int (Char.code @@ Bytes.get m (k + 2))) 8 lor
           (Int32.of_int (Char.code @@ Bytes.get m (k + 3)))
done;
```

16.1.6 Looping over blocks

⟨Sha1.sha1() loop from 0 to 79 for each block i 111c⟩≡ (110b)

```
for t = 0 to 79 do
  let f, k =
    if t <= 19 then (!b land !c) lor ((lnot !b) land !d), 0x5A827999l else
    if t <= 39 then !b lxor !c lxor !d, 0x6ED9EBA1l else
    if t <= 59 then
      (!b land !c) lor (!b land !d) lor (!c land !d), 0x8F1BBCDCl
    else
```

```
      !b lxor !c lxor !d, 0xCA62C1D6l
    in
    let s = t &&& 0xF in
    if (t >= 16) then begin
      w.(s) <- cls 1 begin
          w.((s + 13) &&& 0xF) lxor
          w.((s + 8) &&& 0xF) lxor
          w.((s + 2) &&& 0xF) lxor
          w.(s)
        end
    end;
    let temp = (cls 5 !a) ++ f ++ !e ++ w.(s) ++ k in
    e := !d;
    d := !c;
    c := cls 30 !b;
    b := !a;
    a := temp;
  done;
```

16.1.7 Padding

⟨*function* sha_1_pad 112a⟩≡ (110b)

```
  let sha_1_pad s =
    let len = String.length s in
    let blen = 8 * len in
    let rem = len mod 64 in
    let mlen = if rem > 55 then len + 128 - rem else len + 64 - rem in
    let m = Bytes.create mlen in
    Bytes.blit_string s 0 m 0 len;
    Bytes.fill m len (mlen - len) '\x00';
    Bytes.set m len '\x80';
    if Sys.word_size > 32 then begin
      Bytes.set m (mlen - 8) (Char.unsafe_chr (blen lsr 56 land 0xFF));
      Bytes.set m (mlen - 7) (Char.unsafe_chr (blen lsr 48 land 0xFF));
      Bytes.set m (mlen - 6) (Char.unsafe_chr (blen lsr 40 land 0xFF));
      Bytes.set m (mlen - 5) (Char.unsafe_chr (blen lsr 32 land 0xFF));
    end;
    Bytes.set m (mlen - 4) (Char.unsafe_chr (blen lsr 24 land 0xFF));
    Bytes.set m (mlen - 3) (Char.unsafe_chr (blen lsr 16 land 0xFF));
    Bytes.set m (mlen - 2) (Char.unsafe_chr (blen lsr 8 land 0xFF));
    Bytes.set m (mlen - 1) (Char.unsafe_chr (blen land 0xFF));
    m
  in
```

16.2 Unzip

⟨*signature* Unzip.inflate 112b⟩≡ (184)

```
  val inflate : ?header:bool -> IO.input -> IO.input
  (** wrap an input using "inflate" decompression algorithm. raises [Error] if
    an error occurs (this can only be caused by malformed input data). *)
```

16.2.1 Overview

16.2.2 Data structures

⟨*type* Unzip.huffman 112c⟩≡ (185)

```
  type huffman =
    (* Leaf *)
    | Found of int (** 0..288 *)
    (* Node *)
    | NeedBit of huffman * huffman
    ⟨Unzip.huffman cases 124e⟩
  [@@warning "-37"]
```

Uses Unzip.huffman 112c.

⟨type Unzip.window 113a⟩≡ (185)

```
  type window = {
    mutable wbuffer : bytes;
    mutable wpos : int;
    ⟨Unzip.window other fields 122a⟩
  }
```

⟨constant Unzip.window_size 113b⟩≡ (185)

```
  let window_size = 1 lsl 15
```

⟨constant Unzip.buffer_size 113c⟩≡ (185)

```
  let buffer_size = 1 lsl 16
```

Uses Unzip.window.wpos 113a.

⟨function Unzip.window_create 113d⟩≡ (185)

```
  let window_create () = {
      wbuffer = Bytes.create buffer_size;
      wpos = 0;
      ⟨Unzip.window_create() set other fields 122b⟩
    }
```

Uses Unzip.adler32_update() 122c, Unzip.window.wbuffer 113a, Unzip.window.wcrc, and Unzip.window_size.

⟨type Unzip.state 113e⟩≡ (185)

```
  type state =
    | Head

    | Block
    | CData

    | Done
    ⟨Unzip.state other cases 121c⟩
  [@@warning "-37"]
```

⟨type Unzip.t 113f⟩≡ (185)

```
  type t = {
    mutable zstate    : state;

    mutable zhuffman  : huffman;
    zwindow  : window;

    (* input *)
    zinput    : IO.input;

    (* output *)
    mutable zoutput   : bytes;
    mutable zoutpos   : int;
    mutable zneeded   : int;

    ⟨Unzip.t other fields 114a⟩
  }
```

Uses IO.input, Unzip.huffman 112c, and Unzip.window.

⟨Unzip.t *other fields* 114a⟩≡ (113f) 114e▷
```
mutable zfinal    : bool;
```

16.2.3 Error management

⟨*type* Unzip.error_msg 114b⟩≡ (185 184)
```
type error_msg =
  | Invalid_huffman
  | Invalid_data
  | Invalid_crc
  | Truncated_data
  | Unsupported_dictionary
```

⟨*exception* Unzip.Error 114c⟩≡ (185 184)
```
exception Error of error_msg
```

⟨*function* Unzip.error 114d⟩≡ (185)
```
let error msg = raise (Error msg)
```

16.2.4 IO helpers

Input bits helpers

⟨Unzip.t *other fields* 114e⟩+≡ (113f) ◁114a 120b▷
```
(** usually a byte, but can contained more *)
mutable zbits     : int;
(** unread (not consumed) bits in zbits (usually 0..8, but can be more) *)
mutable znbits    : int;
```

⟨*function* Unzip.reset_bits 114f⟩≡ (185)
```
let reset_bits z =
  z.zbits <- 0;
  z.znbits <- 0
```
Uses Unzip.t.zwindow 113f and Unzip.window_add_bytes() 117a.

⟨*function* Unzip.get_bit 114g⟩≡ (185)
```
let get_bit z =
  if z.znbits = 0 then begin
    z.znbits <- 8;
    z.zbits <- IO.read_byte z.zinput;
  end;
  let b = z.zbits land 1 = 1 in
  z.znbits <- z.znbits - 1;
  z.zbits <- z.zbits lsr 1;
  b
```
Uses Unzip.t.zbits and Unzip.t.znbits 114e.

⟨*function* Unzip.get_bits 114h⟩≡ (185)
```
let get_bits z n =
  while z.znbits < n do
    z.zbits <- z.zbits lor ((IO.read_byte z.zinput) lsl z.znbits);
    z.znbits <- z.znbits + 8;
  done;
  let b = z.zbits land (1 lsl n - 1) in
  z.znbits <- z.znbits - n;
  z.zbits <- z.zbits lsr n;
  b
```
Uses Unzip.t.zbits and Unzip.t.znbits 114e.

Output helpers

⟨*function* `Unzip.add_bytes` 115a⟩≡ (185)
```
let add_bytes z s p l =
  ⟨Unzip.add_bytes() add bytes to window 116d⟩
  Bytes.unsafe_blit s p z.zoutput z.zoutpos l;
  z.zneeded <- z.zneeded - l;
  z.zoutpos <- z.zoutpos + l
```
Uses `Unzip.t.zoutpos` 113f, `Unzip.t.zwindow` 113f, and `Unzip.window_add_char()`.

⟨*function* `Unzip.add_char` 115b⟩≡ (185)
```
let add_char z c =
  ⟨Unzip.add_char() add character to window 116f⟩
  Bytes.unsafe_set z.zoutput z.zoutpos c;
  z.zneeded <- z.zneeded - 1;
  z.zoutpos <- z.zoutpos + 1
```
Uses `Unzip.t.zoutpos` 113f, `Unzip.t.zwindow` 113f, and `Unzip.window_get_last_char()`.

16.2.5 Huffman trees

⟨*function* `Unzip.apply_huffman` 115c⟩≡ (185)
```
let rec apply_huffman z = function
  | Found n -> n
  | NeedBit (a,b) -> apply_huffman z (if get_bit z then b else a)
 ⟨Unzip.apply_huffman() other cases 124f⟩
```

⟨*constant* `Unzip.fixed_huffman` 115d⟩≡ (185)
```
let fixed_huffman =
  make_huffman (Array.init 288 (fun n ->
    if n <= 143 then 8
    else if n <= 255 then 9
    else if n <= 279 then 7
    else 8
  )) 0 288 10
```
Uses `IO.read_byte()`, `Unzip.t.zbits`, `Unzip.t.zinput` 113f, and `Unzip.t.znbits` 114e.

⟨*function* `Unzip.make_huffman` 115e⟩≡ (185)
```
let make_huffman lengths pos nlengths maxbits =

  let counts = Array.make maxbits 0 in
  for i = 0 to nlengths - 1 do
    let p = Array.unsafe_get lengths (i + pos) in
    ⟨Unzip.make_huffman() sanity check codelength p 116c⟩
    Array.unsafe_set counts p (Array.unsafe_get counts p + 1);
  done;

  let code = ref 0 in
  let tmp = Array.make maxbits 0 in
  for i = 1 to maxbits - 2 do
    code := (!code + Array.unsafe_get counts i) lsl 1;
    Array.unsafe_set tmp i !code;
  done;

  let bits = Hashtbl.create 0 in
  for i = 0 to nlengths - 1 do
    let l = Array.unsafe_get lengths (i + pos) in
    if l <> 0 then begin
      let n = Array.unsafe_get tmp (l - 1) in
      Array.unsafe_set tmp (l - 1) (n + 1);
```

```
        Hashtbl.add bits (n,l) i;
      end;
    done;

    ⟨function Unzip.tree_make 116a⟩
      (NeedBit (tree_make 0 1, tree_make 1 1))
```

Uses Unzip.error() 185 and Unzip.error_msg.Invalid_huffman.

⟨function Unzip.tree_make 116a⟩≡ (115e)
```
let rec tree_make v l =
  ⟨Unzip.tree_make() sanity check l 116b⟩
  try
    Found (Hashtbl.find bits (v,l))
  with
    Not_found ->
      NeedBit (tree_make (v lsl 1) (l + 1) , tree_make (v lsl 1 lor 1) (l + 1))
in
```

Uses Unzip.huffman.NeedBit 112c.

⟨Unzip.tree_make() sanity check l 116b⟩≡ (116a)
```
if l > maxbits then error Invalid_huffman;
```

Uses Unzip.huffman.NeedBit 112c.

⟨Unzip.make_huffman() sanity check codelength p 116c⟩≡ (115e)
```
if p >= maxbits then error Invalid_huffman;
```

16.2.6 Sliding window back references (LZ77)

⟨Unzip.add_bytes() add bytes to window 116d⟩≡ (115a)
```
window_add_bytes z.zwindow s p l;
```

⟨function Unzip.window_add_bytes 116e⟩≡ (185)
```
let window_add_bytes w s p len =
  ⟨Unzip.window_add_bytes() slide window if reached end of buffer 116i⟩
  Bytes.unsafe_blit s p w.wbuffer w.wpos len;
  w.wpos <- w.wpos + len
```

Uses Unzip.buffer_size 185 and Unzip.window.wpos 113a.

⟨Unzip.add_char() add character to window 116f⟩≡ (115b)
```
window_add_char z.zwindow c;
```

⟨function Unzip.window_add_char 116g⟩≡ (185)
```
let window_add_char w c =
  ⟨Unzip.window_add_char() slide window if reached end of buffer 116h⟩
  Bytes.unsafe_set w.wbuffer w.wpos c;
  w.wpos <- w.wpos + 1
```

Uses Unzip.window.wbuffer 113a and Unzip.window.wpos 113a.

⟨Unzip.window_add_char() slide window if reached end of buffer 116h⟩≡ (116g)
```
if w.wpos = buffer_size
then window_slide w;
```

⟨Unzip.window_add_bytes() slide window if reached end of buffer 116i⟩≡ (116e)
```
if w.wpos + len > buffer_size
then window_slide w;
```

⟨*function* `Unzip.window_slide` 117a⟩≡ (185)
```
let window_slide w =
  ⟨Unzip.window_slide() update crc before blit 122d⟩
  let b = Bytes.create buffer_size in
  w.wpos <- w.wpos - window_size;
  Bytes.unsafe_blit w.wbuffer window_size b 0 w.wpos;
  w.wbuffer <- b
```
Uses `Unzip.buffer_size` 185, `Unzip.window.wbuffer` 113a, `Unzip.window.wpos` 113a, and `Unzip.window_size`.

⟨*function* `Unzip.window_available` 117b⟩≡ (185)
```
let window_available w =
  w.wpos
```
Uses `Unzip.adler32_update()` 122c, `Unzip.window.wbuffer` 113a, `Unzip.window.wcrc`, and `Unzip.window.wpos` 113a.

16.2.7 Entry point: `inflate()`

⟨*function* `Unzip.inflate` 117c⟩≡ (185)
```
let inflate ?(header=true) ch =
  let z = inflate_init ~header ch in
  let tmp = Bytes.create 1 in
  (* read input close *)
  IO.create_in
    (* special case of ~input for 1 byte *)
    (*~read:*)(fun() ->
      let l = inflate_data z tmp 0 1 in
      if l = 1
      then Bytes.unsafe_get tmp 0
      else raise IO.No_more_input
    )
    (*~input:*)(fun buf_dst pos_in_buf len ->
      let n = inflate_data z buf_dst pos_in_buf len in
      if n = 0
      then raise IO.No_more_input;
      n
    )
    (*~close:*)(fun () ->
      IO.close_in ch
    )
```
Uses `IO.close_in()` and `Unzip.inflate_data()` 125d.

⟨*signature* `Unzip.inflate_init` 117d⟩≡ (184)
```
val inflate_init : ?header:bool -> IO.input -> t
```

⟨*function* `Unzip.inflate_init` 117e⟩≡ (185)
```
let inflate_init ?(header=true) ch =
  {
    zstate = (if header then Head else Block);
    zinput = ch;
    zfinal = false;

    zoutput = Bytes.empty;
    zoutpos = 0;
    zneeded = 0;

    zhuffman = fixed_huffman;
    zhuffdist = None;

    zwindow = window_create ();
```

```
    zbits = 0;
    znbits = 0;

    zlen = 0;
    zdist = 0;
    zlengths = Array.make 19 (-1);
  }
```

Uses Unzip.fixed_huffman, Unzip.t.zbits, Unzip.t.zdist 120b, Unzip.t.zhuffdist 123e, Unzip.t.zhuffman 113f, Unzip.t.zlen 114e, Unzip.t.zlengths 120c, Unzip.t.znbits 114e, Unzip.t.zneeded 113f, Unzip.t.zoutpos 113f, Unzip.t.zoutput 113f, Unzip.t.zwindow 113f, and Unzip.window_create().

⟨*signature* Unzip.inflate_data 118a⟩≡ (184)
```
val inflate_data : t -> bytes -> int -> int -> int
```

⟨*function* Unzip.inflate_data 118b⟩≡ (185)
```
let inflate_data z buf_dst pos_in_buf len =
  ⟨Unzip.inflate_data() sanity check parameters 118c⟩
  z.zneeded <- len;
  z.zoutpos <- pos_in_buf;
  z.zoutput <- buf_dst;
  try
    if len > 0
    then inflate_loop z;
    len - z.zneeded
  with IO.No_more_input -> error Truncated_data
```

Uses Unzip.error() 185, Unzip.error_msg.Truncated_data 114b, Unzip.state.Block 113e, Unzip.state.Head 113e, Unzip.t.zneeded 113f, Unzip.t.zoutpos 113f, and Unzip.t.zstate.

⟨Unzip.inflate_data() *sanity check parameters* 118c⟩≡ (118b)
```
if pos_in_buf < 0 || len < 0 || pos_in_buf + len > Bytes.length buf_dst
then invalid_arg "inflate_data";
```

16.2.8 inflate_loop()

⟨*function* Unzip.inflate_loop 118d⟩≡ (185)
```
let rec inflate_loop z =
  match z.zstate with
  ⟨Unzip.inflate_loop() match state cases 118e⟩
```

Head

⟨Unzip.inflate_loop() *match state cases* 118e⟩≡ (118d) 119b▷
```
| Head ->
  let cmf = IO.read_byte z.zinput in
  ⟨Unzip.inflate_loop() when in Head state, sanity check cmf 118f⟩
  let flg = IO.read_byte z.zinput in
  ⟨Unzip.inflate_loop() when in Head state, sanity check flg 119a⟩
  z.zstate <- Block;
  inflate_loop z
```

Uses Unzip.error() 185, Unzip.error_msg.Invalid_data 114b, and Unzip.error_msg.Unsupported_dictionary 114b.

⟨Unzip.inflate_loop() *when in Head state, sanity check* cmf 118f⟩≡ (118e)
```
let cm    = cmf land 15 in
let cinfo = cmf lsr 4 in
if cm <> 8 || cinfo <> 7
then error Invalid_data;
```

Uses Unzip.error() 185 and Unzip.error_msg.Invalid_data 114b.

⟨Unzip.inflate_loop() *when in Head state, sanity check* flg 119a⟩≡ (118e)
```
let fdict = flg land 32 <> 0 in
if (cmf lsl 8 + flg) mod 31 <> 0
then error Invalid_data;
if fdict
then error Unsupported_dictionary;
```
Uses Unzip.get_bit() 114h, Unzip.inflate_loop() 124d, Unzip.state.Block 113e, Unzip.t.zfinal 113f, and Unzip.t.zstate.

Block

⟨Unzip.inflate_loop() *match state cases* 119b⟩+≡ (118d) ◁118e 119d▷
```
| Block ->
  z.zfinal <- get_bit z;
  let btype = get_bits z 2 in
  (match btype with
  ⟨Unzip.inflate_loop() when in Block state, match block type cases 119c⟩
  | _ ->
    error Invalid_data
  )
```
Uses Unzip.add_char() 115a, Unzip.debug 185, Unzip.fixed_huffman, Unzip.state.CData 113e, Unzip.t.zhuffdist 123e, Unzip.t.zhuffman 113f, and Unzip.t.zstate.

⟨Unzip.inflate_loop() *when in Block state, match block type cases* 119c⟩≡ (119b) 123a▷
```
| 1 -> (* fixed Huffman *)
  if !debug then print_string "Unzip: Fixed Huffman\n";
  z.zhuffman <- fixed_huffman;
  z.zhuffdist <- None;
  z.zstate <- CData;
  inflate_loop z
```
Uses IO.LittleEndian.read_ui16(), Unzip.debug 185, Unzip.t.zinput 113f, and Unzip.t.zlen 114e.

CData

⟨Unzip.inflate_loop() *match state cases* 119d⟩+≡ (118d) ◁119b 121d▷
```
| CData ->
  (match apply_huffman z z.zhuffman with
  ⟨Unzip.inflate_loop() when in CData state, match apply huffman cases 119e⟩
  )
```
Uses Unzip.inflate_loop() 124d and Unzip.t.zlen 114e.

⟨Unzip.inflate_loop() *when in CData state, match apply huffman cases* 119e⟩≡ (119d) 119f▷
```
| n when n < 256 ->
  add_char z (Char.unsafe_chr n);
  if z.zneeded > 0
  then inflate_loop z
```
Uses Unzip.inflate_loop() 124d, Unzip.state.Block 113e, Unzip.state.Crc, Unzip.t.zfinal 113f, and Unzip.t.zstate.

⟨Unzip.inflate_loop() *when in CData state, match apply huffman cases* 119f⟩+≡ (119d) ◁119e 120a▷
```
| 256 ->
  z.zstate <- if z.zfinal then Crc else Block;
  inflate_loop z
```
Uses Unzip.len_extra_bits_tbl 122g.

⟨Unzip.inflate_loop() *when in CData state, match apply huffman cases* 120a⟩+≡ (119d) ◁119f
```
| n ->
  let n = n - 257 in
  ⟨Unzip.inflate_loop() when in CData state, when backref, compute zlen 120d⟩
  ⟨Unzip.inflate_loop() when in CData state, when backref, compute zdist 120g⟩

  if z.zdist > window_available z.zwindow
  then error Invalid_data;
  z.zstate <- Dist;
  inflate_loop z
```
Uses Unzip.get_bits() 115d, Unzip.len_base_val_tbl 185, Unzip.state.Dist, Unzip.t.zdist 120b, Unzip.t.zlen 114e, and Unzip.t.zneeded 113f.

⟨Unzip.t *other fields* 120b⟩+≡ (113f) ◁114e 120c▷
```
mutable zlen      : int;
```

⟨Unzip.t *other fields* 120c⟩+≡ (113f) ◁120b 123e▷
```
mutable zdist     : int;
```

zlen

⟨Unzip.inflate_loop() *when in CData state, when backref, compute zlen* 120d⟩≡ (120a)
```
let extra_bits = Array.unsafe_get len_extra_bits_tbl n in
if extra_bits = -1
then error Invalid_data;
z.zlen <- (Array.unsafe_get len_base_val_tbl n) + (get_bits z extra_bits);
```
Uses Unzip.get_rev_bits() 114g and Unzip.t.zhuffdist 123e.

⟨*constant* Unzip.len_extra_bits_tbl 120e⟩≡ (185)
```
let len_extra_bits_tbl = [|0;0;0;0;0;0;0;0;1;1;1;1;2;2;2;2;3;3;3;3;4;4;4;4;5;5;5;5;0;-1;-1|]
```

⟨*constant* Unzip.len_base_val_tbl 120f⟩≡ (185)
```
let len_base_val_tbl = [|3;4;5;6;7;8;9;10;11;13;15;17;19;23;27;31;35;43;51;59;67;83;99;115;131;163;195;227;258|
```

zdist

⟨Unzip.inflate_loop() *when in CData state, when backref, compute zdist* 120g⟩≡ (120a)
```
let dist_code =
  match z.zhuffdist with
  | None -> get_rev_bits z 5
  ⟨Unzip.inflate_loop() compute dist_code, match zhuffdist cases 124b⟩
in
let extra_bits = Array.unsafe_get dist_extra_bits_tbl dist_code in
if extra_bits = -1
then error Invalid_data;
z.zdist <- (Array.unsafe_get dist_base_val_tbl dist_code) + (get_bits z extra_bits);
```
Uses Unzip.dist_extra_bits_tbl 120e, Unzip.error() 185, Unzip.error_msg.Invalid_data 114b, Unzip.inflate_loop() 124d, Unzip.state.Dist, Unzip.t.zdist 120b, Unzip.t.zstate, Unzip.t.zwindow 113f, and Unzip.window_available().

⟨*function* Unzip.get_rev_bits 120h⟩≡ (185)
```
let rec get_rev_bits z n =
  if n = 0 then
    0
  else if get_bit z then
    (1 lsl (n - 1)) lor (get_rev_bits z (n-1))
  else
    get_rev_bits z (n-1)
```
Uses Unzip.get_rev_bits() 114g, Unzip.t.zbits, and Unzip.t.znbits 114e.

⟨*constant* `Unzip.dist_extra_bits_tbl` 121a⟩≡ (185)
```
let dist_extra_bits_tbl = [|0;0;0;0;1;1;2;2;3;3;4;4;5;5;6;6;7;7;8;8;9;9;10;10;11;11;12;12;13;13;-1;-1|]
```

⟨*constant* `Unzip.dist_base_val_tbl` 121b⟩≡ (185)
```
let dist_base_val_tbl = [|1;2;3;4;5;7;9;13;17;25;33;49;65;97;129;193;257;385;513;769;1025;1537;2049;3073;4097;6
```

Dist

⟨`Unzip.state` *other cases* 121c⟩≡ (113e) 121f▷
```
| Dist
```

⟨`Unzip.inflate_loop()` *match state cases* 121d⟩+≡ (118d) ◁119d 121g▷
```
| Dist ->
  while z.zlen > 0 && z.zneeded > 0 do
    let len = min z.zneeded (min z.zlen z.zdist) in
    add_dist z z.zdist len;
    z.zlen <- z.zlen - len;
  done;
  if z.zlen = 0
  then z.zstate <- CData;
  if z.zneeded > 0
  then inflate_loop z
```
Uses `Unzip.adler32_read()` 122e, `Unzip.inflate_loop()` 124d, `Unzip.state.CData` 113e, `Unzip.state.Crc`, `Unzip.t.zinput` 113f, `Unzip.t.zlen` 114e, `Unzip.t.zneeded` 113f, `Unzip.t.zstate`, `Unzip.t.zwindow` 113f, and `Unzip.window_checksum()`.

⟨*function* `Unzip.add_dist` 121e⟩≡ (185)
```
let add_dist z d l =
  add_bytes z z.zwindow.wbuffer (z.zwindow.wpos - d) l
```
Uses `Unzip.apply_huffman()`, `Unzip.get_bit()` 114h, `Unzip.huffman.Found` 112c, and `Unzip.huffman.NeedBit` 112c.

CRC

⟨`Unzip.state` *other cases* 121f⟩+≡ (113e) ◁121c 123b▷
```
| Crc
```

⟨`Unzip.inflate_loop()` *match state cases* 121g⟩+≡ (118d) ◁121d 121h▷
```
| Crc ->
  ⟨Unzip.inflate_loop() when in Crc state, check adler32 checksum 122f⟩
  z.zstate <- Done;
  inflate_loop z
```
Uses `Unzip.state.Flat`, `Unzip.t.zlen` 114e, and `Unzip.t.zneeded` 113f.

Done

⟨`Unzip.inflate_loop()` *match state cases* 121h⟩+≡ (118d) ◁121g 123c▷
```
| Done ->
  ()
```
Uses `Unzip.t.zlen` 114e.

16.2.9 Advanced features

Adler32 CRC

⟨*type* `Unzip.adler32` 121i⟩≡ (185)
```
type adler32 = {
  mutable a1 : int;
  mutable a2 : int;
}
```

⟨Unzip.window *other fields* 122a⟩≡ (113a)
```
wcrc : adler32;
```

⟨Unzip.window_create() *set other fields* 122b⟩≡ (113d)
```
wcrc = adler32_create();
```

⟨*function* Unzip.adler32_create 122c⟩≡ (185)
```
let adler32_create() = {
  a1 = 1;
  a2 = 0;
}
```

⟨Unzip.window_slide() *update crc before blit* 122d⟩≡ (117a)
```
adler32_update w.wcrc w.wbuffer 0 window_size;
```

⟨*function* Unzip.adler32_update 122e⟩≡ (185)
```
let adler32_update a s p l =
  let p = ref p in
  for _i = 0 to l - 1 do
    let c = int_of_char (Bytes.unsafe_get s !p) in
    a.a1 <- (a.a1 + c) mod 65521;
    a.a2 <- (a.a2 + a.a1) mod 65521;
    incr p;
  done
```
Uses IO.read_byte().

⟨Unzip.inflate_loop() *when in Crc state, check adler32 checksum* 122f⟩≡ (121g)
```
let calc = window_checksum z.zwindow in
let crc = adler32_read z.zinput in
if calc <> crc
then error Invalid_crc;
```
Uses Unzip.inflate_loop() 124d and Unzip.state.Done 113e.

⟨*function* Unzip.window_checksum 122g⟩≡ (185)
```
let window_checksum w =
  adler32_update w.wcrc w.wbuffer 0 w.wpos;
  w.wcrc
```

⟨*function* Unzip.adler32_read 122h⟩≡ (185)
```
let adler32_read ch =
  let a2a = IO.read_byte ch in
  let a2b = IO.read_byte ch in
  let a1a = IO.read_byte ch in
  let a1b = IO.read_byte ch in
  {
    a1 = (a1a lsl 8) lor a1b;
    a2 = (a2a lsl 8) lor a2b;
  }
```
Uses Unzip.adler32.a1 121i and Unzip.adler32.a2 121i.

No compression block type

⟨Unzip.inflate_loop() *when in Block state, match block type cases* 123a⟩+≡ (119b) ◁119c 123d▷

```
| 0 -> (* no compression *)
  if !debug then print_string "Unzip: no compression\n";
  z.zlen <- IO.LittleEndian.read_ui16 z.zinput;
  let nlen = IO.LittleEndian.read_ui16 z.zinput in
  if nlen <> 0xffff - z.zlen
  then error Invalid_data;
  z.zstate <- Flat;
  inflate_loop z;
  reset_bits z
```

Uses Unzip.debug 185, Unzip.get_bits() 115d, Unzip.inflate_loop() 124d, Unzip.reset_bits() 120h, Unzip.state.Flat, and Unzip.t.zstate.

⟨Unzip.state *other cases* 123b⟩+≡ (113e) ◁121f 125c▷

```
| Flat
```

⟨Unzip.inflate_loop() *match state cases* 123c⟩+≡ (118d) ◁121h 125d▷

```
| Flat ->
  let rlen = min z.zlen z.zneeded in
  let str = IO.nread z.zinput rlen in
  let len = Bytes.length str in
  z.zlen <- z.zlen - len;
  add_bytes z str 0 len;
  if z.zlen = 0
  then z.zstate <- (if z.zfinal then Crc else Block);
  if z.zneeded > 0
  then inflate_loop z
```

Uses Unzip.add_dist_one() 115b, Unzip.debug 185, Unzip.inflate_loop() 124d, Unzip.state.Block 113e, Unzip.state.Crc, Unzip.state.DistOne, Unzip.t.zfinal 113f, Unzip.t.zlen 114e, Unzip.t.zneeded 113f, and Unzip.t.zstate.

Specialized huffman tree

⟨Unzip.inflate_loop() *when in Block state, match block type cases* 123d⟩+≡ (119b) ◁123a

```
| 2 -> (* dynamic Huffman *)
  if !debug then print_string "Unzip: Dynamic Huffman\n";
  let hlit = get_bits z 5 + 257 in
  let hdist = get_bits z 5 + 1 in
  let hclen = get_bits z 4 + 4 in
  for i = 0 to hclen - 1 do
    Array.unsafe_set z.zlengths (Array.unsafe_get code_lengths_pos i) (get_bits z 3);
  done;
  for i = hclen to 18 do
    Array.unsafe_set z.zlengths (Array.unsafe_get code_lengths_pos i) 0;
  done;
  z.zhuffman <- make_huffman z.zlengths 0 19 8;
  let lengths = Array.make (hlit + hdist) 0 in
  inflate_lengths z lengths (hlit + hdist);
  z.zhuffdist <- Some (make_huffman lengths hlit hdist 16);
  z.zhuffman <- make_huffman lengths 0 hlit 16;
  z.zstate <- CData;
  inflate_loop z
```

Uses Unzip.code_lengths_pos 121a, Unzip.error() 185, Unzip.error_msg.Invalid_data 114b, Unzip.get_bits() 115d, Unzip.inflate_lengths(), Unzip.inflate_loop() 124d, Unzip.make_huffman() 125a, Unzip.state.CData 113e, Unzip.t.zhuffdist 123e, Unzip.t.zhuffman 113f, Unzip.t.zlengths 120c, and Unzip.t.zstate.

⟨Unzip.t *other fields* 123e⟩+≡ (113f) ◁120c 124a▷

```
zlengths : int array;
```

Uses Unzip.huffman 112c.

⟨Unzip.t *other fields* 124a⟩+≡ (113f) ◁123e
```
mutable zhuffdist : huffman option;
```

⟨Unzip.inflate_loop() *compute* dist_code, *match* zhuffdist *cases* 124b⟩≡ (120g)
```
| Some h ->
  apply_huffman z h
```
Uses Unzip.dist_base_val_tbl 120f, Unzip.error() 185, Unzip.error_msg.Invalid_data 114b, Unzip.get_bits() 115d, and Unzip.t.zdist 120b.

⟨*constant* Unzip.code_lengths_pos 124c⟩≡ (185)
```
let code_lengths_pos = [|16;17;18;0;8;7;9;6;10;5;11;4;12;3;13;2;14;1;15|]
```

⟨*function* Unzip.inflate_lengths 124d⟩≡ (185)
```
let inflate_lengths z a max =
  let i = ref 0 in
  let prev = ref 0 in
  while !i < max do
    match apply_huffman z z.zhuffman with
    | n when n <= 15 ->
      prev := n;
      Array.unsafe_set a !i n;
      incr i
    | 16 ->
      let n = 3 + get_bits z 2 in
      if !i + n > max then error Invalid_data;
      for _k = 0 to n - 1 do
        Array.unsafe_set a !i !prev;
        incr i;
      done;
    | 17 ->
      let n = 3 + get_bits z 3 in
      i := !i + n;
      if !i > max then error Invalid_data;
    | 18 ->
      let n = 11 + get_bits z 7 in
      i := !i + n;
      if !i > max then error Invalid_data;
    | _ ->
      error Invalid_data
  done
```
Uses Unzip.error() 185, Unzip.error_msg.Invalid_data 114b, Unzip.get_bits() 115d, and Unzip.t.zstate.

16.2.10 Optimizations

NeedBits

⟨Unzip.huffman *cases* 124e⟩≡ (112c)
```
(* Opti *)
| NeedBits of int * huffman array
```
Uses Unzip.huffman 112c.

⟨Unzip.apply_huffman() *other cases* 124f⟩≡ (115c)
```
| NeedBits (n,t) -> apply_huffman z (Array.unsafe_get t (get_bits z n))
```

⟨*function* Unzip.tree_compress 125a⟩≡ (185)

```
(*
let rec _tree_compress t =
  match tree_depth t with
  | 0 -> t
  | 1 ->
    (match t with
    | NeedBit (a,b) -> NeedBit (tree_compress a,tree_compress b)
    | _ -> assert false)
  | d ->
    let size = 1 lsl d in
    let tbl = Array.make size (Found (-1)) in
    tree_walk tbl 0 0 d t;
    NeedBits (d,tbl)

and _tree_walk tbl p cd d = function
  | NeedBit (a,b) when d > 0 ->
    tree_walk tbl p (cd + 1) (d-1) a;
    tree_walk tbl (p lor (1 lsl cd)) (cd + 1) (d-1) b;
  | t ->
    Array.set tbl p (tree_compress t)
*)
```

Uses Unzip.huffman.Found 112c, Unzip.huffman.NeedBit 112c, Unzip.huffman.NeedBits 124e, Unzip.tree_compress() 125b, and Unzip.tree_walk() 125a.

⟨*function* Unzip.tree_depth 125b⟩≡ (185)

```
(*
let rec tree_depth = function
  | Found _ -> 0
  | NeedBit (a,b) ->
    1 + min (tree_depth a) (tree_depth b)
  | NeedBits _ -> assert false
*)
```

Uses Unzip.huffman.NeedBit 112c, Unzip.huffman.NeedBits 124e, and Unzip.tree_depth() 185.

DistOne

⟨Unzip.state *other cases* 125c⟩+≡ (113e) ◁123b

```
| DistOne
```

⟨Unzip.inflate_loop() *match state cases* 125d⟩+≡ (118d) ◁123c

```
| DistOne ->
  if !debug then print_string "Unzip: DistOne\n";
  let len = min z.zlen z.zneeded in
  add_dist_one z len;
  z.zlen <- z.zlen - len;
  if z.zlen = 0
  then z.zstate <- CData;
  if z.zneeded > 0
  then inflate_loop z
```

Uses Unzip.inflate_loop() 124d, Unzip.state.CData 113e, Unzip.t.zneeded 113f, and Unzip.t.zstate.

⟨*function* Unzip.add_dist_one 125e⟩≡ (185)

```
let add_dist_one z n =
  let c = window_get_last_char z.zwindow in
  let s = Bytes.make n c in
  add_bytes z s 0 n
```

Uses Unzip.add_bytes() 114f, Unzip.t.zwindow 113f, Unzip.window.wbuffer 113a, and Unzip.window.wpos 113a.

⟨*function* `Unzip.window_get_last_char` 126a⟩≡ (185)
```
let window_get_last_char w =
  Bytes.unsafe_get w.wbuffer (w.wpos - 1)
```
Uses `Unzip.window.wpos` 113a.

16.3 Zip

16.3.1 Entry point: `deflate()`

⟨*function* `Zip.deflate` 126b⟩≡
```
TODO lpize zip.ml, and also port to OCaml zip algorithm
```

⟨*signature* `Zlib.compress` 126c⟩≡ (188c)
```
val compress:
  ?level: int -> ?header: bool ->
  (bytes -> int) -> (bytes -> int -> unit) -> unit
```

⟨*function* `Zlib.compress` 126d⟩≡ (189)
```
let compress ?(level = 6) ?(header = true) refill flush =
  let inbuf = Bytes.create buffer_size
  and outbuf = Bytes.create buffer_size in
  let zs = deflate_init level header in
  let rec compr inpos inavail =
    if inavail = 0 then begin
      let incount = refill inbuf in
      if incount = 0 then compr_finish() else compr 0 incount
    end else begin
      let (_, used_in, used_out) =
        deflate zs inbuf inpos inavail outbuf 0 buffer_size Z_NO_FLUSH in
      flush outbuf used_out;
      compr (inpos + used_in) (inavail - used_in)
    end
  and compr_finish () =
    let (finished, _, used_out) =
       deflate zs inbuf 0 0 outbuf 0 buffer_size Z_FINISH in
    flush outbuf used_out;
    if not finished then compr_finish()
  in
    compr 0 0;
    deflate_end zs
```
Uses `Zlib.buffer_size` 127a, `Zlib.deflate()` 189, `Zlib.deflate_end()` 189, `Zlib.deflate_init()` 189, `Zlib.flush_command.Z_FINISH` 126g, and `Zlib.flush_command.Z_NO_FLUSH` 126g.

⟨*exception* `Zlib.Error` 126e⟩≡ (189 188c)
```
exception Error of string * string
```

⟨*toplevel* `Zlib._1` 126f⟩≡ (189)
```
let _ =
  Callback.register_exception "Zlib.Error" (Error("",""))
```

⟨*type* `Zlib.flush_command` 126g⟩≡ (189 188c)
```
type flush_command =
    Z_NO_FLUSH
  | Z_SYNC_FLUSH
  | Z_FULL_FLUSH
  | Z_FINISH
```

⟨*constant* Zlib.buffer_size 127a⟩≡ (189)
```
let buffer_size = 1024
```

⟨*signature* Zlib.compress_direct 127b⟩≡ (188c)
```
val compress_direct:
  ?level: int -> ?header: bool -> (bytes -> int -> unit) ->
  (bytes -> int -> int -> unit) * (unit -> unit)
```

⟨*function* Zlib.compress_direct 127c⟩≡ (189)
```
let compress_direct  ?(level = 6) ?(header = true) flush =
  let outbuf = Bytes.create buffer_size in
  let zs = deflate_init level header in
  let rec compr inbuf inpos inavail =
    if inavail = 0 then ()
    else begin
      let (_, used_in, used_out) =
        deflate zs inbuf inpos inavail outbuf 0 buffer_size Z_NO_FLUSH in
      flush outbuf used_out;
      compr inbuf (inpos + used_in) (inavail - used_in)
    end
  and compr_finish () =
    let (finished, _, used_out) =
      deflate zs (Bytes.unsafe_of_string "") 0 0
                 outbuf 0 buffer_size Z_FINISH in
    flush outbuf used_out;
    if not finished then compr_finish()
  in
  compr, compr_finish
```
Uses Zlib.buffer_size 127a, Zlib.deflate() 189, Zlib.deflate_init() 189, Zlib.flush_command.Z_FINISH 126g, and Zlib.flush_command.Z_NO_FLUSH 126g.

⟨*signature* Zlib.uncompress 127d⟩≡ (188c)
```
val uncompress:
  ?header: bool -> (bytes -> int) -> (bytes -> int -> unit) -> unit
```

⟨*function* Zlib.uncompress 127e⟩≡ (189)
```
let uncompress ?(header = true) refill flush =
  let inbuf = Bytes.create buffer_size
  and outbuf = Bytes.create buffer_size in
  let zs = inflate_init header in
  let rec uncompr inpos inavail =
    if inavail = 0 then begin
      let incount = refill inbuf in
      if incount = 0 then uncompr_finish true else uncompr 0 incount
    end else begin
      let (finished, used_in, used_out) =
        inflate zs inbuf inpos inavail outbuf 0 buffer_size Z_SYNC_FLUSH in
      flush outbuf used_out;
      if not finished then uncompr (inpos + used_in) (inavail - used_in)
    end
  and uncompr_finish first_finish =
    (* Gotcha: if there is no header, inflate requires an extra "dummy" byte
       after the compressed stream in order to complete decompression
       and return finished = true. *)
    let dummy_byte = if first_finish && not header then 1 else 0 in
    let (finished, _, used_out) =
       inflate zs inbuf 0 dummy_byte outbuf 0 buffer_size Z_SYNC_FLUSH in
    flush outbuf used_out;
    if not finished then uncompr_finish false
  in
```

```
    uncompr 0 0;
    inflate_end zs
```

Uses Zlib.buffer_size 127a, Zlib.flush_command.Z_SYNC_FLUSH 126g, Zlib.inflate() 189, Zlib.inflate_end() 189, and Zlib.inflate_init() 189.

16.4 Diff

16.4.1 Overview

16.4.2 Data structures

16.4.3 Entry point: Diffs.diff()

⟨*signature* Diff_basic.diff 128a⟩≡ (161b)

```
val diff: string array -> string array -> Diff.diff
```

⟨*signature* Diff_myers.diff 128b⟩≡ (163a)

```
val diff: string array -> string array -> Diff.diff
```

⟨*function* Diffs.diff 128c⟩≡ (166b)

```
let diff str1 str2 =
  let xs = split_lines str1 in
  let ys = split_lines str2 in
  Diff_myers.diff (Array.of_list xs) (Array.of_list ys)
```

Uses Diffs.split_lines() 128d.

⟨*function* Diffs.split_lines 128d⟩≡ (166b)

```
let split_lines str =
  (* alt: let xs = Str.full_split (Str.regexp "\n") str in *)
  let rec aux start =
    try
      let idx = String.index_from str start '\n' in
      let line = String.sub str start (idx - start + 1) in
      line::aux (idx + 1)
    with Not_found ->
      if start = String.length str
      then []
      else [String.sub str start (String.length str - start)]
  in
  aux 0
```

16.4.4 Edit distance basic algorithm (O(n^2) in space/time)

⟨*function* Diff_basic.diff 128e⟩≡ (161c)

```
let diff arr1 arr2 =
  (* opti: string to int *)

  let mat = matrix_distance arr1 arr2 in
  let trace = traceback_transcript arr1 arr2 mat in
  List.rev trace

  (* opti: get back string from int *)
(*
  let arr1, arr2, revh = hash_strings arr1 arr2 in
  |> List.map (function
      | Diff.Added i   -> Diff.Added   (revh.(i))
```

```
      | Diff.Deleted i -> Diff.Deleted (revh.(i))
      | Diff.Equal i   -> Diff.Equal   (revh.(i))
  )
 *)
```

Uses Diff_basic.matrix_distance() 129a and Diff_basic.traceback_transcript() 129b.

⟨*function* Diff_basic.matrix_distance 129a⟩≡ (161c)

```
 let matrix_distance arr1 arr2 =
   let n = Array.length arr1 in
   let m = Array.length arr2 in
   (* this can be big ... *)
   let mat = Array.make_matrix (n+1) (m+1) 0 in
   let t i j =
     if Array.get arr1 (i-1) = Array.get arr2 (j-1)
     then 0
     else 1
   in
   let min3 a b c = min (min a b) c in

   begin
     for i = 0 to n do
       mat.(i).(0) <- i
     done;
     for j = 0 to m do
       mat.(0).(j) <- j;
     done;
     (* this can be long ... *)
     for i = 1 to n do
       for j = 1 to m do
         mat.(i).(j) <-
           min3 (mat.(i).(j-1) + 1)
                (mat.(i-1).(j) + 1)
                (mat.(i-1).(j-1) + t i j)
       done
     done;
     mat
   end
```

⟨*function* Diff_basic.traceback_transcript 129b⟩≡ (161c)

```
 (* extract the traceback from the matrice (Gusfield P221) *)
 let traceback_transcript arr1 arr2 mat =
   let n = Array.length arr1 in
   let m = Array.length arr2 in
   let get_orig_arr arr i = arr.(i-1) in
   (* you need Figure 11.3 P222 of Gusfield book to understand the code below *)
   let rec aux i j =
     let x = mat.(i).(j) in
     match () with
     | _ when i = 0 && j = 0 -> []
     | _ when i = 0 -> (Diff.Added (get_orig_arr arr2 j))::aux (i) (j-1)
     | _ when j = 0 -> (Diff.Deleted (get_orig_arr arr1 i))::aux (i-1) (j)
     | _ when x = mat.(i-1).(j) + 1 ->
       (Diff.Deleted (get_orig_arr arr1 i))::aux (i-1) (j)
     | _ when x = mat.(i).(j-1) + 1 ->
       (Diff.Added (get_orig_arr arr2 j))::aux (i) (j-1)
     | _ ->
       if x = mat.(i-1).(j-1)
       then Diff.Equal (get_orig_arr arr1 (i))::aux (i-1) (j-1)
       else (Diff.Deleted (get_orig_arr arr1 i))::
            (Diff.Added (get_orig_arr arr2 j))::
```

```
            aux (i-1) (j-1)
    in
    aux n m
```

Uses Diff.diff_elem.Added 85b, Diff.diff_elem.Deleted 85b, and Diff.diff_elem.Equal.

16.4.5 Myers algorithm (O(n) in space in best-case)

⟨*function* Diff_myers.lcs 130a⟩≡ (163b)

```
  let lcs a b =
    let n = Array.length a in
    let m = Array.length b in
    let mn = m + n in
    let sz = 2 * mn + 1 in

    let vd = Array.make sz 0 in
    let vl = Array.make sz 0 in
    let vr = Array.make sz [] in

    let get v i = v.(i + mn) in
    let set v i x = v.(i + mn) <- x in
    ⟨function Diff_myers.finish 131a⟩
    if mn = 0
    then []
    else
      (* For d <- 0 to mn Do *)
      let rec dloop d =
        assert (d <= mn);
        (* For k <- -d to d in steps of 2 Do *)
        ⟨function Diff_myers.kloop 130b⟩
        in kloop (-d)
      in dloop 0
```

⟨*function* Diff_myers.kloop 130b⟩≡ (130a)

```
  let rec kloop k =
    if k > d
    then dloop (d + 1)
    else
      let x, l, r =
        if k = -d || (k <> d && get vd (k - 1) < get vd (k + 1))
        then get vd (k + 1), get vl (k + 1), get vr (k + 1)
        else get vd (k - 1) + 1, get vl (k - 1), get vr (k - 1)
      in
      let x, y, l, r =
        ⟨function Diff_myers.xyloop 130c⟩
        xyloop x (x - k) l r
      in
      set vd k x;
      set vl k l;
      set vr k r;
      if x >= n && y >= m
      then
        (* Stop *)
        finish ()
      else
        kloop (k + 2)
```

⟨*function* Diff_myers.xyloop 130c⟩≡ (130b)

```
  let rec xyloop x y l r =
    if x < n && y < m && equal a.(x) b.(y)
```

```
    then xyloop (x + 1) (y + 1) (l + 1) (‘Common(x, y, a.(x))::r)
    else x, y, l, r
  in
```

Uses `Diff_myers.equal()` 163b.

⟨*function* `Diff_myers.finish` 131a⟩≡ (130a)

```
  let finish () =
    let rec loop i maxl r =
      match () with
      | _ when i > mn -> List.rev r
      | _ when get vl i > maxl -> loop (i + 1) (get vl i) (get vr i)
      | _ -> loop (i + 1) maxl r
    in loop (- mn) 0 []
  in
```

⟨*function* `Diff_myers.diff` 131b⟩≡ (163b)

```
  let diff a b =
    let append_map g arr from to_ init =
      let rec loop i init =
        if i >= to_
        then init
        else loop (i + 1) (g i (arr.(i)):: init)
      in loop from init
    in
    let added   _i x = Diff.Added x in
    let removed _i x = Diff.Deleted x in
    let rec loop cs apos bpos init =
      match cs with
      | [] ->
          init
          |> append_map removed a apos (Array.length a)
          |> append_map added   b bpos (Array.length b)
      | ‘Common (aoff, boff, x) :: rest ->
          init
          |> append_map removed a apos aoff
          |> append_map added   b bpos boff
          |> (fun y -> (Diff.Equal x)::y)
          |> loop rest (aoff + 1) (boff + 1)
    in loop (lcs a b) 0 0 [] |> List.rev
```

Uses `Diff.diff_elem.Added` 85b, `Diff.diff_elem.Deleted` 85b, `Diff.diff_elem.Equal`, and `Diff_myers.lcs()` 130a.

16.5 Diff3

16.5.1 Overview

16.5.2 Data structures

16.5.3 Entry point: `diff3()`

⟨*function* `Diff3.diff3` 131c⟩≡

```
  TODO
```

16.5.4 Displaying merging conflicts

Chapter 17

Advanced Features TODO

17.1 Tracking symbolic links

⟨Tree.perm *cases* 132a⟩≡ (36d) 132e▷
```
| Link
```

⟨Index.mode *cases* 132b⟩≡ (39e) 132k▷
```
| Link
```

⟨Index.stat_info_of_lstats() *match kind and perm cases* 132c⟩≡ (40h)
```
| Unix.S_LNK, _ -> Link
```

⟨Repository.content_from_path_and_unix_stat() *match kind cases* 132d⟩≡ (64a)
```
| Unix.S_LNK ->
  Unix.readlink !!full_path
```

17.2 Splitting a large repository: submodules

⟨Tree.perm *cases* 132e⟩+≡ (36d) ◁132a
```
| Commit (* ?? submodule? *)
```

⟨Tree.perm_of_string() *match str cases* 132f⟩≡ (43d)
```
| "160000" -> Commit
```
Uses Common.spf().

⟨Tree.string_of_perm() *match perm cases* 132g⟩≡ (51g)
```
| Commit -> "160000"
```

⟨Repository.set_worktree_and_index_to_tree() *walk tree cases* 132h⟩≡ (77d)
```
| Tree.Commit -> failwith "submodule not yet supported"
```
Uses Tree.perm.Commit 132a.

⟨Tree.walk_tree() *match perm cases* 132i⟩≡ (78b)
```
| Commit ->
  failwith "submodule not supported yet"
```

⟨Repository.build_file_from_blob() *match perm cases* 132j⟩≡ (78c)
```
| Tree.Commit -> failwith "submodule not yet supported"
```
Uses Tree.perm.Commit 132a.

⟨Index.mode *cases* 132k⟩+≡ (39e) ◁132b
```
| Gitlink (*?? submodule? *)
```

⟨Index.read_mode() *match* n lsr 12 *cases* 133a⟩≡ (48c)
```
| 0b1110 -> Gitlink
```

⟨Index.write_mode() *match mode cases* 133b⟩≡ (54a)
```
| Gitlink -> 0b1110__000__000_000_000
```
Uses Index.stat_info.ctime 39d and Index.write_time() 173b.

⟨Changes.content_from_path_and_stat_index() *match mode cases* 133c⟩≡ (86b)
```
| Index.Gitlink -> failwith "submodule not supported"
```

⟨Index.perm_of_mode() *match mode cases* 133d⟩≡ (69d)
```
| Gitlink -> Tree.Commit (* sure? *)
```

⟨Index.mode_of_perm() *match perm cases* 133e⟩≡ (93c)
```
| Tree.Commit -> Gitlink
```
Uses Tree.perm.Dir 36d.

17.3 Configuration file: .git/config

17.3.1 Initialization

17.3.2 Config.t

17.3.3 Reading a configuration file

17.3.4 Writing a configuration file

17.3.5 Stacked configurations: /.gitconfig and .git/config

17.4 Commit rules and checks: .git/hooks/

17.5 Ignoring files: .gitignore

17.6 Archeology: Grafts

17.6.1 Reading a graft file

17.6.2 Writing a graft file

17.6.3 Using a graft file

17.7 Rename detection

Chapter 18

Advanced Commands TODO

18.1 Tagging a commit with a name

18.1.1 `Tag.t`

⟨`Objects.t` *cases* 134a⟩≡ (35d)
```
(*  | Tag of Tag.t *)
```

⟨`Repository.objectish` *cases* 134b⟩+≡ (38f) ◁??
```
(* ObjByTag *)
```

18.1.2 Reading a tag

⟨`Objects.read()` *match str cases* 134c⟩+≡ (41d) ◁43g
```
(* "tag" -> Tag (Tag.read raw) *)
```

18.1.3 Writing a tag

⟨`Objects.write()` *match obj cases* 134d⟩≡ (50a)

⟨`Objects.write()` *return header, match obj cases* 134e⟩≡ (50a)

⟨`Cmd_show.show()` *match obj cases* 134f⟩+≡ (82b) ◁83d

Chapter 19

Advanced Networking TODO

19.1 Other clients

19.1.1 ssh://

⟨Clients.client_of_url() *match url cases* 137a⟩+≡ (99c) ◁108a 137b▷

```
  | s when s =~ "^ssh://" ->
    failwith "ssh not supported"
```

19.1.2 http://

⟨Clients.client_of_url() *match url cases* 137b⟩+≡ (99c) ◁137a

```
  | s when s =~ "^http://" ->
    failwith "http not supported"
```

19.2 Other servers

19.2.1 http://

19.3 Other client/server capabilities

19.3.1 Report status

19.3.2 Quiet

19.3.3 Thin pack

19.3.4 Side band 64k

Chapter 20

Advanced Topics TODO

20.1 Bytes versus ASCII versus UTF-8

20.2 Reliability and signals

20.3 Optimizations

20.4 Fast import and export

20.5 Alternates

20.6 Other repository format

20.6.1 Bare repository

20.6.2 `.git` file

20.7 Security

Chapter 21

Conclusion

Appendix A

Debugging

⟨*constant* Cmds.extra_commands 140a⟩≡ (158)
```
let extra_commands = [
  Cmd_test.cmd;
  Cmd_dump.cmd;
]
```
Uses Cmd_dump.cmd 140b and Cmd_test.cmd 141g.

A.1 ocamlgit dump

⟨*constant* Cmd_dump.cmd 140b⟩≡ (154d)
```
let cmd = { Cmd_.
  name = "dump";
  usage = " <file>";
  options = [
    "-raw", Arg.Set raw, " do not pretty print";
    "-index", Arg.Set index, " pretty print index content";
  ];
  f = (fun args ->
    match args with
    | [file] -> dump file
    | _ -> failwith (spf "dump command [%s] not supported"
                         (String.concat ";" args))
  );
}
```
Uses Cmd.t.f 55a, Cmd.t.name 55a, Cmd.t.options 55a, Cmd.t.usage 55a, Cmd_dump.dump() 140e, Cmd_dump.index 140d, Cmd_dump.raw 140c, and Common.spf().

⟨*constant* Cmd_dump.raw 140c⟩≡ (154d)
```
let raw = ref false
```

⟨*constant* Cmd_dump.index 140d⟩≡ (154d)
```
let index = ref false
```

⟨*function* Cmd_dump.dump 140e⟩≡ (154d)
```
let dump file =
  if !index
  then dump_index file
  else dump_object file
```
Uses Cmd_dump.dump_index() 141a, Cmd_dump.dump_object() 141d, and Cmd_dump.index 140d.

⟨*function* Cmd_dump.dump_index 141a⟩≡ (154d)
```
(* =~ dulwich dump-index, =~ git ls-files --stage *)
let dump_index file =
  let chan = open_in file in
  let input = IO.input_channel chan in
  let index = Index.read input in
  let v = Dump.vof_index index in
  UConsole.print (OCaml.string_of_v v)
```
Uses Common.pr(), Dump.vof_index() 169b, IO.input_channel(), Index.read() 53e, and Ocaml.string_of_v().

⟨*signature* Dump.vof_index 141b⟩≡ (169a)
```
val vof_index: Index.t -> OCaml.v
```

⟨*constant* Dump.vof_index 141c⟩≡ (169b)
```
let vof_index = Index.vof_t
```

⟨*function* Cmd_dump.dump_object 141d⟩≡ (154d)
```
(* =~ git cat-file -p *)
let dump_object file =
  let chan = open_in file in
  let input = IO.input_channel chan in
  let unzipped = Unzip.inflate input in

  try
    if !raw
    then
      let str = IO.read_all unzipped in
      Logs.app (fun m -> m "%s" str)
    else begin
      let obj = Objects.read unzipped in
      let v = Dump.vof_obj obj in
      UConsole.print (OCaml.string_of_v v)
    end
  with Unzip.Error _err ->
    failwith "unzip error"
```
Uses Cmd_dump.raw 140c, Common.pr(), Common.pr2(), Dump.vof_obj() 169b, IO.input_channel(), IO.read_all(), Objects.read() 41d, Ocaml.string_of_v(), and Unzip.inflate() 185.

⟨*signature* Dump.vof_obj 141e⟩≡ (169a)
```
val vof_obj: Objects.t -> OCaml.v
```

⟨*constant* Dump.vof_obj 141f⟩≡ (169b)
```
let vof_obj = Objects.vof_t
```

A.2 ocamlgit test

⟨*constant* Cmd_test.cmd 141g⟩≡ (157)
```
let cmd = { Cmd_.
  name = "test";
  usage = " ";
  options = [];
  f = (fun args ->
    match args with
    | ["sha1"; str] -> test_sha1 str
    | ["sha1"] -> test_sha1 "what is up, doc?"

    | ["diff";file1;file2] -> test_diff file1 file2
    | ["diff"] ->
```

```
      failwith "missing arguments to diff (diff <file1> <file2>)"

    | ["diff3";file1;file2;file3] -> test_diff3 file1 file2 file3
    | ["diff3"] ->
      failwith "missing arguments to diff3 (diff3 <orig> <filea> <fileb>)"

    | ["unzip"; file] -> test_unzip file
    | ["zip"; file] -> test_zip file

    | ["unzip_all_objects"] -> test_unzip_all_objects ()

    | _ -> failwith (spf "test command [%s] not supported"
                         (String.concat ";" args))
  );
}
```

Uses Cmd.t.f 55a, Cmd.t.name 55a, Cmd.t.options 55a, Cmd.t.usage 55a, Cmd_test.test_diff() 157, Cmd_test.test_diff3() 157, Cmd_test.test_sha1() 142, Cmd_test.test_unzip() 157, Cmd_test.test_unzip_all_objects() 157, Cmd_test.test_zip() 157, and Common.spf().

⟨*function* Cmd_test.test_sha1 142⟩≡ (157)

```
(* see https://git-scm.com/book/en/v2/Git-Internals-Git-Objects *)
let test_sha1 content =
  let header = spf "blob %d\000" (String.length content) in
  let store = header ^ content in

  let sha = Sha1.sha1 store in
  UConsole.print (spf "len = %d, raw = %s" (String.length sha) sha);
  let hexsha = Hexsha.of_sha sha in
  UConsole.print (spf "len = %d, str = %s" (String.length hexsha) hexsha);
  ()
```

Uses Common.spf().

Appendix B

Profiling

⟨Main.main() *GC settings* 143⟩≡ (56b)

```
Gc.set {(Gc.get ()) with Gc.stack_limit = 1000 * 1024 * 1024};
```

Appendix C

Error Management

Appendix D

Utilities

D.1 Common functions

D.2 File utilities

D.3 Directory utilities

⟨*signature* `Repository.walk_dir` 145a⟩≡ (179)

```
val walk_dir:
  (Fpath.t (* dir *) -> string (* subdir *) list -> string (* file *) list -> unit) ->
  Fpath.t ->
  unit
```

⟨*function* `Repository.walk_dir` 145b⟩≡ (180)

```
(* inspired from os.path.walk in Python *)
let rec walk_dir f (dir : Fpath.t) : unit =
  dir |> with_opendir (fun handle ->
    let dirs = ref [] in
    let files = ref [] in
    try
      while true do
        let s = Unix.readdir handle in
        (* git specific here *)
        if s <> "." && s <> ".." && s <> ".git" then begin
          let path : Fpath.t = dir / s in
          let st = Unix.lstat !!path in
          (match st.Unix.st_kind with
          | Unix.S_DIR -> Stack_.push s dirs
          | _ -> Stack_.push s files
          )
        end
      done
    with End_of_file ->
      let dirs = List.rev !dirs in
      let files = List.rev !files in
      f dir dirs files;
      dirs |> List.iter (fun s ->
        walk_dir f (dir / s)
      )
  )
```

Uses `Common.push()`, `Repository.walk_dir()` 145b, and `Repository.with_opendir()` 146a.

⟨*function* `Repository.with_opendir` 146a⟩≡ (180)
```
(* less: use finalize *)
let with_opendir f (dir : Fpath.t) =
  let handle = Unix.opendir !!dir in
  let res = f handle in
  Unix.closedir handle;
  res
```

D.4 Generalized IO

D.5 Binary IO

⟨*signature* `IO_.with_close_out` 146b⟩≡ (148a)
```
val with_close_out: ('a IO.output -> unit) -> 'a IO.output -> 'a
```

⟨*function* `IO_.with_close_out` 146c⟩≡ (148b)
```
let with_close_out f ch =
  f ch;
  let res = IO.close_out ch in
  res
```
Uses `IO.close_out()`.

⟨*signature* `IO_.read_string_and_stop_char` 146d⟩≡ (148a)
```
val read_string_and_stop_char:
  IO.input -> char -> string
```

⟨*function* `IO_.read_string_and_stop_char` 146e⟩≡ (148b)
```
let read_string_and_stop_char ch stop_char =
  let b = Buffer.create 8 in
  let rec loop() =
    let c = IO.read ch in
    if c <> stop_char then begin
      Buffer.add_char b c;
      loop();
    end;
  in
  loop();
  Buffer.contents b
```
Uses `IO.read()`.

⟨*signature* `IO_.read_int_and_nullbyte` 146f⟩≡ (148a)
```
val read_int_and_nullbyte:
  IO.input -> int
```

⟨*function* `IO_.read_int_and_nullbyte` 146g⟩≡ (148b)
```
let read_int_and_nullbyte ch =
  let str = IO.read_c_string ch in
  int_of_string str
```
Uses `IO.read_c_string()`.

⟨*signature* `IO_.read_key_space_value_newline` 146h⟩≡ (148a)
```
val read_key_space_value_newline:
  IO.input -> string (* key *) -> (IO.input -> 'a) -> 'a
```

⟨*function* `IO_.read_key_space_value_newline` 147⟩≡ (148b)

```
let read_key_space_value_newline ch k f =
  let str = read_string_and_stop_char ch ' ' in
  if str <> k
  then failwith (spf
    "read_key_space_value_newline: wrong key got %s (expecting %s)"
    str k);
  let v = f ch in
  let c = IO.read ch in
  if c <> '\n'
  then failwith "read_key_space_value_newline: wrong format, no newline";
  v
```

Uses `Common.spf()`, `IO.read()`, and `IO_.read_string_and_stop_char()` 146e.

Appendix E

Extra Code

E.1 version_control/

E.1.1 version_control/IO_.mli

⟨version_control/IO_.mli 148a⟩≡

⟨*signature* IO_.read_string_and_stop_char 146d⟩

⟨*signature* IO_.read_int_and_nullbyte 146f⟩

⟨*signature* IO_.read_key_space_value_newline 146h⟩

⟨*signature* IO_.with_close_out 146b⟩

E.1.2 version_control/IO_.ml

⟨version_control/IO_.ml 148b⟩≡

```
⟨copyright ocamlgit 15a⟩
open Common

(*****************************************************************************)
(* Prelude *)
(*****************************************************************************)

(*****************************************************************************)
(* Entry points *)
(*****************************************************************************)
```

⟨*function* IO_.with_close_out 146c⟩

⟨*function* IO_.read_string_and_stop_char 146e⟩

⟨*function* IO_.read_int_and_nullbyte 146g⟩

⟨*function* IO_.read_key_space_value_newline 147⟩

E.1.3 version_control/blob.mli

⟨version_control/blob.mli 148c⟩≡

⟨*type* Blob.t 35e⟩

⟨*type* `Blob.hash` 35f⟩

⟨*signature* `Blob.read` 42d⟩
⟨*signature* `Blob.write` 50e⟩

⟨*signature* `Blob.show` 82d⟩

E.1.4 version_control/blob.ml

⟨`version_control/blob.ml` 149a⟩≡
⟨*copyright ocamlgit* 15a⟩

```
(*****************************************************************************)
(* Prelude *)
(*****************************************************************************)

(*****************************************************************************)
(* Types *)
(*****************************************************************************)
```

⟨*type* `Blob.t` 35e⟩

⟨*type* `Blob.hash` 35f⟩

```
(*****************************************************************************)
(* IO *)
(*****************************************************************************)
```

⟨*function* `Blob.read` 42e⟩

⟨*function* `Blob.write` 51a⟩

```
(*****************************************************************************)
(* Show *)
(*****************************************************************************)
```

⟨*function* `Blob.show` 82e⟩

E.1.5 version_control/change.mli

⟨`version_control/change.mli` 149b⟩≡

⟨*type* `Change.content` 84f⟩

⟨*type* `Change.entry` 84e⟩

⟨*type* `Change.t` 84d⟩

E.1.6 version_control/change.ml

⟨`version_control/change.ml` 149c⟩≡
⟨*copyright ocamlgit* 15a⟩

```
(*****************************************************************************)
(* Prelude *)
(*****************************************************************************)
```

```
(*****************************************************************************)
(* Types *)
(*****************************************************************************)
```
⟨*type* `Change.content` 84f⟩

⟨*type* `Change.entry` 84e⟩

⟨*type* `Change.t` 84d⟩

E.1.7 version_control/changes.mli

⟨`version_control/changes.mli` 150a⟩≡

⟨*signature* `Changes.changes_tree_vs_tree` 91b⟩

⟨*signature* `Changes.changes_worktree_vs_index` 85f⟩

⟨*signature* `Changes.changes_index_vs_tree` 95d⟩

E.1.8 version_control/changes.ml

⟨`version_control/changes.ml` 150b⟩≡

⟨*copyright ocamlgit* 15a⟩

```
open Common
open Fpath_.Operators

(*****************************************************************************)
(* Prelude *)
(*****************************************************************************)
(*
 * todo:
 *  - rename detection
 *)

(*****************************************************************************)
(* Types *)
(*****************************************************************************)

(*****************************************************************************)
(* Helpers *)
(*****************************************************************************)
```

⟨*function* `Changes.skip_tree_and_adjust_path` 93a⟩

⟨*function* `Changes.content_from_path_and_stat_index` 86b⟩

```
(*****************************************************************************)
(* Entry points *)
(*****************************************************************************)
```

⟨*function* `Changes.changes_tree_vs_tree` 91c⟩

⟨*function* `Changes.changes_worktree_vs_index` 86a⟩

⟨*function* `Changes.changes_index_vs_tree` 95e⟩

E.1.9 version_control/client.mli

⟨version_control/client.mli 151a⟩≡

⟨*type* Client.t 99a⟩

E.1.10 version_control/client.ml

⟨version_control/client.ml 151b⟩≡
⟨*copyright ocamlgit* 15a⟩

```
(*****************************************************************************)
(* Prelude *)
(*****************************************************************************)

(*****************************************************************************)
(* Types *)
(*****************************************************************************)
```

⟨*type* Client.t 99a⟩

E.1.11 version_control/client_git.mli

⟨version_control/client_git.mli 151c⟩≡

⟨*signature* Client_git.mk_client 108b⟩

E.1.12 version_control/client_git.ml

⟨version_control/client_git.ml 151d⟩≡
⟨*copyright ocamlgit* 15a⟩

```
open Common

(*****************************************************************************)
(* Prelude *)
(*****************************************************************************)

(*****************************************************************************)
(* Types *)
(*****************************************************************************)

(*****************************************************************************)
(* Helpers *)
(*****************************************************************************)

(*****************************************************************************)
(* Entry point *)
(*****************************************************************************)
```

⟨*function* Client_git.mk_client 108c⟩

E.1.13 version_control/client_local.mli

⟨version_control/client_local.mli 151e⟩≡

⟨*signature* Client_local.mk_client 100d⟩

E.1.14 version_control/client_local.ml

⟨version_control/client_local.ml 152a⟩≡
```
  ⟨copyright ocamlgit 15a⟩
  open Common
  open Fpath_.Operators
  open Regexp_.Operators

  (*****************************************************************************)
  (* Prelude *)
  (*****************************************************************************)

  (*****************************************************************************)
  (* Graph walkers *)
  (*****************************************************************************)

  ⟨type Client_local.graph_walker 104a⟩

  ⟨function Client_local.ml_graph_walker 104b⟩

  ⟨function Client_local.collect_filetree 103a⟩

  (*****************************************************************************)
  (* Helpers *)
  (*****************************************************************************)

  ⟨function Client_local.find_top_common_commits 101c⟩

  ⟨function Client_local.iter_missing_objects 102c⟩


  ⟨function Client_local.fetch_objects 101b⟩

  (*****************************************************************************)
  (* Entry point *)
  (*****************************************************************************)

  ⟨function Client_local.mk_client 101a⟩
```

E.1.15 version_control/clients.mli

⟨version_control/clients.mli 152b⟩≡
```
  ⟨signature Clients.client_of_url 99b⟩
```

E.1.16 version_control/clients.ml

⟨version_control/clients.ml 152c⟩≡
```
  ⟨copyright ocamlgit 15a⟩
  open Common
  open Regexp_.Operators

  (*****************************************************************************)
  (* Prelude *)
  (*****************************************************************************)

  (*****************************************************************************)
  (* Entry point *)
```

```
(*****************************************************************************)
```

⟨*function* `Clients.client_of_url` 99c⟩

E.1.17 version_control/cmd_.ml

⟨`version_control/cmd_.ml` 153a⟩≡

⟨*type* `Cmd_.t` 55a⟩

⟨*exception* `Cmd_.ShowUsage` 57c⟩

E.1.18 version_control/cmd_add.ml

⟨`version_control/cmd_add.ml` 153b⟩≡
⟨*copyright ocamlgit* 15a⟩

⟨*function* `Cmd_add.add` 62b⟩

⟨*constant* `Cmd_add.cmd` 62a⟩

E.1.19 version_control/cmd_branch.ml

⟨`version_control/cmd_branch.ml` 153c⟩≡
⟨*copyright ocamlgit* 15a⟩
```
open Common
open Regexp_.Operators
```

⟨*function* `Cmd_branch.list_branches` 73c⟩

⟨*function* `Cmd_branch.create_branch` 75a⟩

⟨*function* `Cmd_branch.delete_branch` 75i⟩

```
(* less: rename_branch *)
```

⟨*constant* `Cmd_branch.del_flag` 75d⟩
⟨*constant* `Cmd_branch.del_force` 75h⟩

⟨*constant* `Cmd_branch.cmd` 73a⟩
Uses `Cmd.t.name` 55a.

E.1.20 version_control/cmd_checkout.ml

⟨`version_control/cmd_checkout.ml` 153d⟩≡
⟨*copyright ocamlgit* 15a⟩
```
open Common
```

⟨*function* `Cmd_checkout.checkout` 76e⟩

⟨*function* `Cmd_checkout.update` 77b⟩

⟨*constant* `Cmd_checkout.cmd` 76c⟩

E.1.21 version_control/cmd_clone.ml

⟨version_control/cmd_clone.ml 154a⟩≡
  ⟨*copyright ocamlgit* 15a⟩

  ⟨*function* Cmd_clone.clone 106b⟩

```
(* todo: when clone then repo should have a "refs/remotes/origin/master" *)
```

  ⟨*constant* Cmd_clone.cmd 106a⟩

E.1.22 version_control/cmd_commit.ml

⟨version_control/cmd_commit.ml 154b⟩≡
  ⟨*copyright ocamlgit* 15a⟩
```
open Common
```

  ⟨*function* Cmd_commit.commit 66b⟩

  ⟨*constant* Cmd_commit.author 70a⟩
  ⟨*constant* Cmd_commit.committer 70e⟩
  ⟨*constant* Cmd_commit.message 70h⟩

  ⟨*constant* Cmd_commit.cmd 66a⟩

E.1.23 version_control/cmd_diff.ml

⟨version_control/cmd_diff.ml 154c⟩≡
  ⟨*copyright ocamlgit* 15a⟩

  ⟨*function* Cmd_diff.diff_worktree_vs_index 85e⟩

  ⟨*constant* Cmd_diff.cmd 85d⟩

E.1.24 version_control/cmd_dump.ml

⟨version_control/cmd_dump.ml 154d⟩≡
  ⟨*copyright ocamlgit* 15a⟩
```
open Common
```

  ⟨*constant* Cmd_dump.raw 140c⟩
  ⟨*constant* Cmd_dump.index 140d⟩

  ⟨*function* Cmd_dump.dump_object 141d⟩

  ⟨*function* Cmd_dump.dump_index 141a⟩

  ⟨*function* Cmd_dump.dump 140e⟩

  ⟨*constant* Cmd_dump.cmd 140b⟩

E.1.25 version_control/cmd_help.ml

⟨version_control/cmd_help.ml 154e⟩≡
  ⟨*copyright ocamlgit* 15a⟩
```
open Common
```

⟨*constant* `Cmd_help.list_extra` 57e⟩

⟨*constant* `Cmd_help.cmd` 58⟩

E.1.26 version_control/cmd_init.ml

⟨`version_control/cmd_init.ml` 155a⟩≡
⟨*copyright ocamlgit* 15a⟩

```
(* less: let bare = ref false *)
```

⟨*constant* `Cmd_init.cmd` 59a⟩

E.1.27 version_control/cmd_log.ml

⟨`version_control/cmd_log.ml` 155b⟩≡
⟨*copyright ocamlgit* 15a⟩

```
open Common
open Fpath_.Operators

(* todo: git log --graph --oneline --decorate --all *)
```

⟨*function* `Cmd_log.print_commit` 90b⟩

⟨*function* `Cmd_log.print_change` 91a⟩

⟨*constant* `Cmd_log.name_status` 90c⟩

⟨*function* `Cmd_log.log` 90a⟩

⟨*constant* `Cmd_log.cmd` 89b⟩

E.1.28 version_control/cmd_merge.ml

⟨`version_control/cmd_merge.ml` 155c⟩≡

E.1.29 version_control/cmd_pull.ml

⟨`version_control/cmd_pull.ml` 155d⟩≡
⟨*copyright ocamlgit* 15a⟩

```
open Common
```

⟨*function* `Cmd_pull.pull` 100a⟩

⟨*constant* `Cmd_pull.cmd` 99d⟩

E.1.30 version_control/cmd_push.ml

⟨`version_control/cmd_push.ml` 155e⟩≡
⟨*copyright ocamlgit* 15a⟩

```
open Common
open Fpath_.Operators
```

⟨*function* Cmd_push.push 105b⟩

⟨*constant* Cmd_push.cmd 105a⟩

E.1.31 version_control/cmd_reset.ml

⟨version_control/cmd_reset.ml 156a⟩≡
⟨*copyright ocamlgit* 15a⟩
open Common

⟨*function* Cmd_reset.reset_hard 80d⟩

⟨*constant* Cmd_reset.hard 79c⟩
⟨*constant* Cmd_reset.soft 80a⟩
⟨*constant* Cmd_reset.mixed 80b⟩

⟨*constant* Cmd_reset.cmd 79a⟩

E.1.32 version_control/cmd_rm.ml

⟨version_control/cmd_rm.ml 156b⟩≡
⟨*copyright ocamlgit* 15a⟩

⟨*function* Cmd_rm.rm 64c⟩

⟨*constant* Cmd_rm.cmd 64b⟩

E.1.33 version_control/cmd_show.ml

⟨version_control/cmd_show.ml 156c⟩≡
⟨*copyright ocamlgit* 15a⟩
open Common

⟨*function* Cmd_show.show 82b⟩

⟨*constant* Cmd_show.cmd 82a⟩

E.1.34 version_control/cmd_status.ml

⟨version_control/cmd_status.ml 156d⟩≡
⟨*copyright ocamlgit* 15a⟩
open Common
open Fpath_.Operators
open Regexp_.Operators

⟨*type* Cmd_status.status 94e⟩

⟨*function* Cmd_status.changes_index_vs_HEAD 95c⟩

⟨*function* Cmd_status.untracked 96⟩

⟨*function* Cmd_status.status_of_repository 95b⟩

⟨*function* Cmd_status.print_change_long 95a⟩

⟨*function* `Cmd_status.print_status_long` 94d⟩

⟨*function* `Cmd_status.print_status_short` 94c⟩

⟨*constant* `Cmd_status.short_format` 94b⟩

⟨*function* `Cmd_status.status` 94a⟩

⟨*constant* `Cmd_status.cmd` 93d⟩

Uses `Cmd.t.name` 55a.

E.1.35 version_control/cmd_test.ml

⟨`version_control/cmd_test.ml` 157⟩≡

```
⟨copyright ocamlgit 15a⟩
open Common
open Fpath_.Operators

(*****************************************************************************)
(* Algorithm tests *)
(*****************************************************************************)

⟨function Cmd_test.test_sha1 142⟩

let test_diff file1 file2 =
  let read_all path =
      let path = Fpath.v path in
      path |> UChan.with_open_in (fun (ch : Chan.i) ->
        ch.ic |> IO.input_channel |> IO.read_all
      )
  in
  let content1 = read_all file1 in
  let content2 = read_all file2 in

  let diffs = Diffs.diff content1 content2 in
  if not (diffs |> List.for_all (function Diff.Equal _ -> true | _ -> false))
  then begin
    UConsole.print (spf "diff --git %s %s" file1 file2);
    (* less: display change of modes *)
    Diff_unified.show_unified_diff diffs
  end

let test_diff3 fileo filea fileb =
  let read_all path =
      let path = Fpath.v path in
      path |> UChan.with_open_in (fun (ch : Chan.i) ->
        ch.ic |> IO.input_channel |> IO.read_all
      )
  in
  let contento = read_all fileo in
  let contenta = read_all filea in
  let contentb = read_all fileb in

  let chunks = Diff3.diff3 contento contenta contentb in
  let str = Diff3.merge filea fileb chunks in
  print_string str
```

```
  let test_unzip file =
    let chan = open_in file in
    let input = IO.input_channel chan in
    let unzipped = Unzip.inflate input in
    let str = IO.read_all unzipped in
    print_string str

  let test_zip file =
    let chan = open_in file in
    let _input = IO.input_channel chan in
    let zipped =
      (* Zip.deflate input *)
      failwith "Zip.deflate"
    in
    let str = IO.read_all zipped in
    UConsole.print str
  (* TODO
    let dst = file ^ ".deflate" in
    let chan = open_out dst in
    let output = IO.output_channel chan in
    IO.nwrite_string output str;
    IO.close_out output
  *)


  let test_unzip_all_objects () =
    Unzip.debug := true;
    let dir = Fpath.v ".git/objects" in
    dir |> Repository.walk_dir (fun path _dirs files ->
      files |> List.iter (fun file ->
        let file = path / file in
        UConsole.print !!file;
        let chan = open_in !!file in
        let input = IO.input_channel chan in
        let unzipped = Unzip.inflate input in
        let _str = IO.read_all unzipped in
        ()
      )
    )

  (*****************************************************************************)
  (* Entry point *)
  (*****************************************************************************)
```

⟨*constant* `Cmd_test.cmd` 141g⟩

Uses `Cmd_test./()` 157, `Common.pr()`, `Common.spf()`, `Common.with_file_in()`, `Diff.diff_elem.Equal`, `Diff3.diff3()`, `Diff3.merge()`, `Diff_unified.show_unified_diff()`, `Diffs.diff()` 128c, `Hexsha.of_sha()`, `IO.input_channel()`, `IO.read_all()`, `Repository.walk_dir()` 145b, `Unzip.debug` 185, `Unzip.inflate()` 185, and `Zip.deflate()`.

E.1.36 version_control/cmds.ml

⟨`version_control/cmds.ml` 158⟩≡

⟨*constant* `Cmds.main_commands` 55b⟩

⟨*constant* `Cmds.extra_commands` 140a⟩

E.1.37 version_control/commit.mli

⟨version_control/commit.mli 159a⟩≡
```
open Common

⟨type Commit.t 36e⟩
⟨type Commit.hash 36f⟩


⟨signature Commit.read 43h⟩
⟨signature Commit.write 51h⟩

⟨signature Commit.show 83e⟩

⟨signature Commit.collect_ancestors 103b⟩

⟨signature Commit.walk_history 88d⟩
```

E.1.38 version_control/commit.ml

⟨version_control/commit.ml 159b⟩≡
```
⟨copyright ocaml-git 15c⟩
open Common

(*****************************************************************************)
(* Prelude *)
(*****************************************************************************)
(*
 * Most of the code below derives from: https://github.com/mirage/ocaml-git
 *)

(*****************************************************************************)
(* Types *)
(*****************************************************************************)

⟨type Commit.t 36e⟩
⟨type Commit.hash 36f⟩

(*****************************************************************************)
(* API *)
(*****************************************************************************)

(* for git log *)

⟨function Commit.walk_history 89a⟩

(* for git pull *)

⟨function Commit.collect_ancestors 103c⟩


(*****************************************************************************)
(* IO *)
(*****************************************************************************)

⟨function Commit.read 44a⟩

⟨function Commit.write 51i⟩
```

```
(*****************************************************************************)
(* Show *)
(*****************************************************************************)
```

⟨*function* `Commit.show` 83f⟩

E.1.39 version_control/compression.mli

⟨`version_control/compression.mli` 160a⟩≡

⟨*signature* `Compression.decompress` 42a⟩
⟨*signature* `Compression.compress` 50b⟩

E.1.40 version_control/compression.ml

⟨`version_control/compression.ml` 160b⟩≡
⟨*copyright ocamlgit* 15a⟩

```
(*****************************************************************************)
(* Prelude *)
(*****************************************************************************)
(* Compression/decompression wrappers.
 *
 * alternatives:
 *  * https://github.com/ygrek/ocaml-extlib/src/unzip.ml
 *    just decompression (inflate), pretty small: 450 LOC
 *  - https://github.com/mirage/decompress
 *    compression/decompression, seems complete, but pretty big.
 *    used by https://github.com/mirage/ocaml-git
 *  * https://github.com/xavierleroy/camlzip
 *    uses C code (some of the code in ocaml-git uses ML code from camlzip)
 *  - https://github.com/madroach/ocaml-zlib
 *    ??
 *  - https://github.com/samoht/ocaml-lz77
 *    copy-pasted in decompress, but too simple. Does not
 *    support right API where can compress/decompress strings.
 *  - libflate in plan9 :)
 *  - ocamlgz from ocamlplot, but just an OCaml binding to C lib
 *
 * Currently used solutions are marked with a '*' above.
 *)

(*****************************************************************************)
(* Entry points *)
(*****************************************************************************)
```

⟨*function* `Compression.decompress` 42b⟩

⟨*function* `Compression.compress` 50c⟩

E.1.41 version_control/diff.mli

⟨`version_control/diff.mli` 160c⟩≡

⟨*type* `Diff.item` 85c⟩

⟨*type* `Diff.diff_elem` 85b⟩

⟨*type* `Diff.diff` 85a⟩

E.1.42 version_control/diff.ml

⟨`version_control/diff.ml` 161a⟩≡
⟨*copyright ocamlgit* 15a⟩

```
(*****************************************************************************)
(* Prelude *)
(*****************************************************************************)

(*****************************************************************************)
(* Types *)
(*****************************************************************************)
```

⟨*type* `Diff.item` 85c⟩

⟨*type* `Diff.diff_elem` 85b⟩

⟨*type* `Diff.diff` 85a⟩

E.1.43 version_control/diff_basic.mli

⟨`version_control/diff_basic.mli` 161b⟩≡

⟨*signature* `Diff_basic.diff` 128a⟩

E.1.44 version_control/diff_basic.ml

⟨`version_control/diff_basic.ml` 161c⟩≡

```
open Common

(*****************************************************************************)
(* Prelude *)
(*****************************************************************************)
(* Basic Diff algorithm based on the computation of the edit distance between
 * two strings (also known as the Longuest Common Subsequence (LCS) problem).
 *
 * reference:
 * "Algorithms on Strings, Trees, and Sequences" by Dan Gusfield, P217.
 *
 * For two strings S1 and S2, D(i,j) is defined to be the edit distance
 * of S1[1..i] to S2[1..j].
 * So the edit distance of S1 (of length n) and S2 (of length m) is D(n,m).
 *
 * Dynamic programming technique
 * base:
 *  D(i,0) = i  for all i
 *   because to go from S1[1..i] to 0 characters of S2 you have to
 *   delete all the characters from S1[1..i]
 *  D(0,j) = j  for all j
 *    because j characters must be inserted
 * recurrence:
```

```
 *  D(i,j) = min([D(i-1, j)+1,
 *                D(i, j-1)+1,
 *                D(i-1, j-1) + t(i,j)])
 *   where t(i,j) is equal to 1 if S1(i) != S2(j) and  0 if equal
 *
 *  Intuition = there are 4 possible actions:
 *    deletion, insertion, substitution, or match
 *  so Lemma =
 *   D(i,j) must be one of the three
 *     D(i, j-1) + 1
 *     D(i-1, j)+1
 *     D(i-1, j-1) +
 *       t(i,j)
 *
 * history: done in summer 2007 for Julia Lawall
 *
 * complexity: O(nm) in time and space, so if you want to diff two
 * sequences of more than 10 000 elts, it will requires a matrix with
 * 100 M and 100 M operations, so this is not very scalable
 * (see Myers's algorithm used in GNU diff for a more scalable algorithm).
 *)

(*****************************************************************************)
(* Algorithm *)
(*****************************************************************************)
```

⟨*function* Diff_basic.matrix_distance 129a⟩

⟨*function* Diff_basic.traceback_transcript 129b⟩

```
(*****************************************************************************)
(* Optimization *)
(*****************************************************************************)
(* Actually this is not a big win. On files of 20 000 LOC, you
 * go from 125s to 118s, so the string comparison in matrix_distance()
 * is not the bottleneck.
 * Actually in native mode you go from 33s to 35s so this is even
 * slowing things down, weird.
 *)
let hash_strings arr1 arr2 =
  let cnt = ref 0 in
  let h = Hashtbl.create (Array.length arr1 / 3) in
  Array.concat [arr1; arr2] |> Array.iter (fun s ->
    if not (Hashtbl.mem h s)
    then begin
      Hashtbl.add h s !cnt;
      incr cnt;
    end
  );
  let revh = Array.make !cnt ("INCORRECT") in
  h |> Hashtbl.iter (fun k v ->
    revh.(v) <- k
  );
  arr1 |> Array.map (fun s -> Hashtbl.find h s),
  arr2 |> Array.map (fun s -> Hashtbl.find h s),
  revh

(* other optimizations:
 *  - strip common lines in the front and end (McIlroy diff was doing that)
 *    so can reduce significantly the size of the matrix
```

```
 *  - Use Hirschberg space opti of storing only the last row.
 *    No need for full matrix to determine the next row (but then
 *    need to be clever to get the traceback).
 *)

(*****************************************************************************)
(* Entry point *)
(*****************************************************************************)
⟨function Diff_basic.diff 128e⟩

(*****************************************************************************)
(* Tests/Bench *)
(*****************************************************************************)
(* good bench: sqlite aggalmation, more than 200 000 LOC
 * time ./ogit test diff /tmp/test1.c /tmp/test2.c
 *  (where test[12].c = sqlite.c or shorter version of sqlite.c)
 *  on my Macbook Air (from around 2014).
 * 200 000 LOC:
 * - Myers: 1.4s (in ocaml bytecode mode)
 * - Basic: ???
 * 20 000 LOC: (matrix_distance requires a matrix of 400M elements)
 * - Myers: 0.1s
 * - Basic: 104s in bytecode, 33s in native
 *)

(*
let edit_distance s1 s2 =
  (matrix_distance s1 s2).(String.length s1).(String.length s2)


let test = edit_distance "vintner" "writers"
let _ = assert (edit_distance "winter" "winter" = 0)
let _ = assert (edit_distance "vintner" "writers" = 5)
*)
```

E.1.45 version_control/diff_myers.mli

⟨version_control/diff_myers.mli 163a⟩≡

```
(**
   An implementation of Eugene Myers' O(ND) Difference Algorithm\[1\].
   This implementation is a port of util.lcs module of
   {{:http://practical-scheme.net/gauche} Gauche Scheme interpreter}.

   - \[1\] Eugene Myers, An O(ND) Difference Algorithm and Its Variations, Algorithmica Vol. 1 No. 2, pp. 251-2
 *)

⟨signature Diff_myers.diff 128b⟩
```

E.1.46 version_control/diff_myers.ml

⟨version_control/diff_myers.ml 163b⟩≡

```
⟨copyright ocaml-diff-myers ??⟩

(* was functorized and parametrized before *)
let equal = (=)

⟨function Diff_myers.lcs 130a⟩
```

⟨*function* `Diff_myers.diff` 131b⟩

E.1.47 version_control/diff_simple.mli

⟨`version_control/diff_simple.mli` 164a⟩≡

```
(** A simple diffing algorithm *)

module type Comparable =
  sig
    type t
      (** The type of the items being compared *)

    val compare: t -> t -> int
      (** A way to distinguish if items are equal or unequal. It follows
          the OCaml convention of returning an integer between -1 to 1. *)
  end

module type S =
  sig
    type item
    (** The type of the item that will be compared. *)

    type diff =
      | Deleted of item array
      | Added of item array
      | Equal of item array
    (** Represents the change or lack of change in a line or character
        between the old and new version. *)

    type t = diff list
    (** List of diffs which is the return value of the main function. *)

    val get_diff : item array -> item array -> t
    (** Returns a list of diffs between two arrays *)
  end

module Make (Item: Comparable) : S with type item = Item.t
```

E.1.48 version_control/diff_simple.ml

⟨`version_control/diff_simple.ml` 164b⟩≡

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

module type S =
  sig
    type item

    type diff =
      | Deleted of item array
      | Added of item array
      | Equal of item array

    type t = diff list
```

```
    val get_diff : item array -> item array -> t
  end

module Make(Item : Comparable) = struct
  type item = Item.t

  type diff =
    | Deleted of item array
    | Added of item array
    | Equal of item array

  type t = diff list

  type subsequence_info =
    { (* Starting index of longest subsequence in the list of new values *)
      sub_start_new : int;
      (* Starting index of longest subsequence in the list of old values *)
      sub_start_old : int;
      (* The length of the longest subsequence *)
      longest_subsequence : int; }

  module CounterMap = Map.Make(Item)

  (* Returns a map with the line as key and a list of indices as value.
     Represents counts of all the lines. *)
  let map_counter keys =
    let keys_and_indices = Array.mapi (fun index key -> index, key) keys in
    Array.fold_left (fun map (index, key) ->
        let indices = try CounterMap.find key map with | Not_found -> [] in
        CounterMap.add key (index :: indices) map
      ) CounterMap.empty keys_and_indices


  (* Computes longest subsequence and returns data on the length of longest
     subsequence and the starting index for the longest subsequence in the old
     and new versions. *)
  let get_longest_subsequence old_lines new_lines =
    let old_values_counter = map_counter old_lines in
    let overlap = Hashtbl.create 5000 in
    let sub_start_old = ref 0 in
    let sub_start_new = ref 0 in
    let longest_subsequence = ref 0 in

    Array.iteri (fun new_index new_value ->
        let indices = try CounterMap.find new_value old_values_counter with
          | Not_found -> []
        in
        List.iter (fun old_index ->
            let prev_subsequence = try Hashtbl.find overlap (old_index - 1) with | Not_found -> 0 in
            let new_subsequence = prev_subsequence + 1 in
            Hashtbl.add overlap old_index new_subsequence;

            if new_subsequence > !longest_subsequence then
              sub_start_old := old_index - new_subsequence + 1;
            sub_start_new := new_index - new_subsequence + 1;
            longest_subsequence := new_subsequence;
          ) indices;
      ) new_lines;
```

```
    { sub_start_new = !sub_start_new;
      sub_start_old = !sub_start_old;
      longest_subsequence = !longest_subsequence }


  let rec get_diff old_lines new_lines =
    match old_lines, new_lines with
    | [||], [||] -> []
    | _, _ ->
      let { sub_start_new; sub_start_old; longest_subsequence } =
        get_longest_subsequence old_lines new_lines
      in

      if longest_subsequence == 0 then
        [Deleted old_lines; Added new_lines]
      else
        let old_lines_presubseq = Array.sub old_lines 0 sub_start_old in
        let new_lines_presubseq = Array.sub new_lines 0 sub_start_new in
        let old_lines_postsubseq =
          let start_index = sub_start_old + longest_subsequence in
          let end_index = Array.length old_lines - start_index in
          Array.sub old_lines start_index end_index
        in
        let new_lines_postsubseq =
          let start_index = sub_start_new + longest_subsequence in
          let end_index = Array.length new_lines - start_index in
          Array.sub new_lines start_index end_index
        in
        let unchanged_lines = Array.sub new_lines sub_start_new longest_subsequence in
        get_diff old_lines_presubseq new_lines_presubseq @
        [Equal unchanged_lines] @
        get_diff old_lines_postsubseq new_lines_postsubseq
end
```

E.1.49 version_control/diffs.mli

⟨version_control/diffs.mli 166a⟩≡

⟨*signature* Diffs.diff 87a⟩

```
(* the split lines contain the trailing '\n' when they have one *)
val split_lines: string -> string list
```

E.1.50 version_control/diffs.ml

⟨version_control/diffs.ml 166b⟩≡
⟨*copyright ocamlgit* 15a⟩

```
(*****************************************************************************)
(* Prelude *)
(*****************************************************************************)
(* Compute the differences between two files line-wise.
 *
 * alternatives:
 *  - Basic diff using simple edit distance algorithm
 *    but O(nm) time and space complexity (space especially is bad)
 *  - Hirshberg invented a linear space optimization of basic diff
```

```
 *  - Hunt and Mcilroy UNIX diff, the original diff program
 *    rely on pair of lines which are the same in both files
 *    (but if the two files contain blank lines, then very bad complexity)
 *  - Myers's diff: https://github.com/leque/ocaml-diff port of
 *    "Eugene Myers, An O(ND) Difference Algorithm and Its Variations,
 *    Algorithmica Vol. 1 No. 2, pp. 251-266, 1986."
 *    https://stackoverflow.com/questions/42635889/myers-diff-algorithm-vs-hunt-mcilroy-algorithm
 *    Apparently Ukkonen discovered independently the same algorithm
 *    Good if the edit distance is small, but very bad if both files are
 *    very different.
 *  - Simple diff: https://github.com/gjaldon/simple-diff
 *    (an OCaml port of https://github.com/paulgb/simplediff )
 *    http://collaboration.cmc.ec.gc.ca/science/rpn/biblio/ddj/Website/articles/DDJ/1988/8807/8807c/8807c.htm a
 *    based on Ratcliff/Obershelp pattern-matching algorithm
 *    seems simple, but not optimal (but "looks right" to people)
 *    but ocaml version seems buggy!
 *  - Bram Cohen's Patience diff: https://github.com/janestreet/patdiff
 *    support also colored output, and word diff, but heavily modularized
 *    https://blog.jcoglan.com/2017/09/19/the-patience-diff-algorithm/
 *    https://blog.jcoglan.com/2017/09/28/implementing-patience-diff/
 *  - Plan9 diff (in plan9/utilities/string/diff/)
 *    not Myers's diff
 *  - GNU diff (in plan9/ape_cmd/diff)
 *    is based on Myers's algorithm
 *  - http://pynash.org/2013/02/26/diff-in-50-lines/
 *    implement Mcilroy in Python (also talk about Python difflib)
 *  - Heckle diff mentionned in diff3.py
 *    "P. Heckel. ``A technique for isolating differences between files.''
 *     Communications of the ACM, Vol. 21, No. 4, page 264, April 1978."
 *
 *)

(*****************************************************************************)
(* Helpers *)
(*****************************************************************************)
⟨function Diffs.split_lines 128d⟩

(*****************************************************************************)
(* Entry point *)
(*****************************************************************************)

⟨function Diffs.diff 128c⟩
```

E.1.51 version_control/diff_unified.mli

⟨version_control/diff_unified.mli 167a⟩≡

```
⟨signature Diff_unified.show_change 87b⟩

(* internals *)
val show_unified_diff: Diff.diff -> unit
```

E.1.52 version_control/diff_unified.ml

⟨version_control/diff_unified.ml 167b⟩≡

```
⟨copyright ocamlgit 15a⟩
open Common
open Fpath_.Operators
```

```
(*****************************************************************************)
(* Prelude *)
(*****************************************************************************)
(* Show differences between 2 files.
 *
 * Short explanation of unified format:
 *  - https://www.gnu.org/software/diffutils/manual/html_node/Detailed-Unified.html
 *  - http://www.artima.com/weblogs/viewpost.jsp?thread=164293
 *
 * alternatives:
 *  - ocamldiff: https://github.com/zoggy/ocamldiff
 *    parse and display unified diffs
 *  - unidiff: https://github.com/gildor478/ocaml-unidiff
 *    parse
 *  - call diff -u (as I did in pfff) directly via Sys.command
 *)

(*****************************************************************************)
(* Helpers *)
(*****************************************************************************)

⟨function Diff_unified.print 88b⟩

⟨function Diff_unified.print_header 88c⟩

⟨constant Diff_unified.nContext 88a⟩

⟨function Diff_unified.show_unified_diff 87d⟩


(*****************************************************************************)
(* Entry points *)
(*****************************************************************************)

⟨function Diff_unified.show_change 87c⟩
```

E.1.53 version_control/diff3.ml

```
⟨version_control/diff3.ml 168⟩≡
  ⟨copyright ocamlgit 15a⟩
  open Common

  (*****************************************************************************)
  (* Prelude *)
  (*****************************************************************************)

  (*
   * alternatives:
   *  - gnu diff3, in C
   *    https://www.gnu.org/software/diffutils/manual/html_node/diff3-Merging.html
   *    apparently by Randy Smith in 1988 and popularized by CVS
   *  - Perl Text::Diff3
   *    http://search.cpan.org/~tociyuki/Text-Diff3-0.10/lib/Text/Diff3.pm
   *    with good example of use in man page
   *  - python diff3
   *    https://github.com/schuhschuh/cmake-basis/blob/master/src/utilities/python/diff3.py
   *    apparently translation of the Perl code, itself a translation of the
   *    C code
```

```
 *  - formal investigation of diff3:
 *    http://www.cis.upenn.edu/~bcpierce/papers/diff3-short.pdf
 *
 *)
```

E.1.54 version_control/dump.mli

⟨version_control/dump.mli 169a⟩≡

```
⟨signature Dump.vof_obj 141e⟩
⟨signature Dump.vof_index 141b⟩
```

E.1.55 version_control/dump.ml

⟨version_control/dump.ml 169b⟩≡

```
open Fpath_.Operators

module Ocaml = OCaml

(* mostly auto-generated by ocamltarzan *)

module Int64 = struct
let vof_t x =
  Ocaml.vof_int (Int64.to_int x)
end

module Int32 = struct
let vof_t x =
  Ocaml.vof_int (Int32.to_int x)
end


module Sha1 = struct
  let vof_t x =
    Ocaml.VSum (("ShaHex", [Ocaml.vof_string (Hexsha.of_sha x)]))
end

module Blob = struct
  let vof_t x = Ocaml.vof_string x
end

module Tree = struct
open Tree
let vof_perm =
  function
  | Normal -> Ocaml.VSum (("Normal", []))
  | Exec -> Ocaml.VSum (("Exec", []))
  | Link -> Ocaml.VSum (("Link", []))
  | Dir -> Ocaml.VSum (("Dir", []))
  | Commit -> Ocaml.VSum (("Commit", []))

let vof_entry { perm = v_perm; name = v_name; id = v_node } =
  let bnds = [] in
  let arg = Sha1.vof_t v_node in
  let bnd = ("id", arg) in
  let bnds = bnd :: bnds in
  let arg = Ocaml.vof_string v_name in
  let bnd = ("name", arg) in
```

```
  let bnds = bnd :: bnds in
  let arg = vof_perm v_perm in
  let bnd = ("perm", arg) in let bnds = bnd :: bnds in Ocaml.VDict bnds

let vof_t v = Ocaml.vof_list vof_entry v
end

module User = struct
open User

let vof_tz_offset x = Ocaml.vof_int x

let vof_t { name = v_name; email = v_email; date = v_date } =
  let bnds = [] in
  let arg =
    match v_date with
    | (v1, v2) ->
        let v1 = Int64.vof_t v1
        and v2 = vof_tz_offset v2
        in Ocaml.VTuple [ v1; v2 ] in
  let bnd = ("date", arg) in
  let bnds = bnd :: bnds in
  let arg = Ocaml.vof_string v_email in
  let bnd = ("email", arg) in
  let bnds = bnd :: bnds in
  let arg = Ocaml.vof_string v_name in
  let bnd = ("name", arg) in let bnds = bnd :: bnds in Ocaml.VDict bnds
end

module Commit = struct
open Commit
let vof_t {
            tree = v_tree;
            parents = v_parents;
            author = v_author;
            committer = v_committer;
            message = v_message
          } =
  let bnds = [] in
  let arg = Ocaml.vof_string v_message in
  let bnd = ("message", arg) in
  let bnds = bnd :: bnds in
  let arg = User.vof_t v_committer in
  let bnd = ("committer", arg) in
  let bnds = bnd :: bnds in
  let arg = User.vof_t v_author in
  let bnd = ("author", arg) in
  let bnds = bnd :: bnds in
  let arg = Ocaml.vof_list Sha1.vof_t v_parents in
  let bnd = ("parents", arg) in
  let bnds = bnd :: bnds in
  let arg = Sha1.vof_t v_tree in
  let bnd = ("tree", arg) in let bnds = bnd :: bnds in Ocaml.VDict bnds
end

module Objects = struct
open Objects
let vof_t =
  function
  | Blob v1 -> let v1 = Blob.vof_t v1 in Ocaml.VSum (("Blob", [ v1 ]))
```

```
  | Commit v1 -> let v1 = Commit.vof_t v1 in Ocaml.VSum (("Commit", [ v1 ]))
  | Tree v1 -> let v1 = Tree.vof_t v1 in Ocaml.VSum (("Tree", [ v1 ]))
end
```
⟨*constant* `Dump.vof_obj` 141f⟩

```
module Index = struct
open Index
let rec
  vof_stat_info {
                  ctime = v_ctime;
                  mtime = v_mtime;
                  dev = v_dev;
                  inode = v_inode;
                  mode = v_mode;
                  uid = v_uid;
                  gid = v_gid;
                  size = v_size
                } =
  let bnds = [] in
  let arg = Int32.vof_t v_size in
  let bnd = ("size", arg) in
  let bnds = bnd :: bnds in
  let arg = Int32.vof_t v_gid in
  let bnd = ("gid", arg) in
  let bnds = bnd :: bnds in
  let arg = Int32.vof_t v_uid in
  let bnd = ("uid", arg) in
  let bnds = bnd :: bnds in
  let arg = vof_mode v_mode in
  let bnd = ("mode", arg) in
  let bnds = bnd :: bnds in
  let arg = Int32.vof_t v_inode in
  let bnd = ("inode", arg) in
  let bnds = bnd :: bnds in
  let arg = Int32.vof_t v_dev in
  let bnd = ("dev", arg) in
  let bnds = bnd :: bnds in
  let arg = vof_time v_mtime in
  let bnd = ("mtime", arg) in
  let bnds = bnd :: bnds in
  let arg = vof_time v_ctime in
  let bnd = ("ctime", arg) in let bnds = bnd :: bnds in Ocaml.VDict bnds
and vof_mode =
  function
  | Normal -> Ocaml.VSum (("Normal", []))
  | Exec -> Ocaml.VSum (("Exec", []))
  | Link -> Ocaml.VSum (("Link", []))
  | Gitlink -> Ocaml.VSum (("Gitlink", []))
and vof_time { lsb32 = v_lsb32; nsec = v_nsec } =
  let bnds = [] in
  let arg = Int32.vof_t v_nsec in
  let bnd = ("nsec", arg) in
  let bnds = bnd :: bnds in
  let arg = Int32.vof_t v_lsb32 in
  let bnd = ("lsb32", arg) in let bnds = bnd :: bnds in Ocaml.VDict bnds


let vof_entry { stats = v_stats; id = v_id; path = v_name }
              =
  let bnds = [] in
```

```
    let arg = Ocaml.vof_string !!v_name in
    let bnd = ("path", arg) in
    let bnds = bnd :: bnds in
    let arg = Sha1.vof_t v_id in
    let bnd = ("id", arg) in
    let bnds = bnd :: bnds in
    let arg = vof_stat_info v_stats in
    let bnd = ("stats", arg) in let bnds = bnd :: bnds in Ocaml.VDict bnds

  let vof_t v = Ocaml.vof_list vof_entry v
  end
```
⟨*constant* `Dump.vof_index` 141c⟩

Uses `Dump.Blob.vof_t()` 169b, `Dump.Commit.vof_t()` 169b, `Dump.Index.vof_entry()` 169b, `Dump.Index.vof_mode()` 169b, `Dump.Index.vof_stat_info()` 169b, `Dump.Index.vof_t()` 169b, `Dump.Index.vof_time()` 169b, `Dump.Int32.vof_t()` 169b, `Dump.Int64.vof_t()` 169b, `Dump.Objects.vof_t()` 169b, `Dump.Sha1.vof_t()` 169b, `Dump.Tree.vof_entry()` 169b, `Dump.Tree.vof_perm()` 169b, `Dump.Tree.vof_t()` 169b, `Dump.User.vof_t()` 169b, `Dump.User.vof_tz_offset()` 169b, `Hexsha.of_sha()`, `Ocaml.v.VDict`, `Ocaml.v.VSum`, `Ocaml.v.VTuple`, `Ocaml.vof_int()`, `Ocaml.vof_list()`, and `Ocaml.vof_string()`.

E.1.56 version_control/hexsha.mli

⟨`version_control/hexsha.mli` 172a⟩≡

```
⟨type Hexsha.t 33a⟩

⟨signature Hexsha.of_sha 34c⟩
⟨signature Hexsha.to_sha 33d⟩

⟨signature Hexsha.read 44b⟩
⟨signature Hexsha.write 52a⟩

⟨signature Hexsha.is_hexsha 33b⟩
```

E.1.57 version_control/hexsha.ml

⟨`version_control/hexsha.ml` 172b⟩≡

```
⟨copyright ocaml-hex ??⟩
open Common
open Regexp_.Operators

(*****************************************************************************)
(* Prelude *)
(*****************************************************************************)
(*
 * Most of the code below comes from src: https://github.com/mirage/ocaml-hex
 *)

(*****************************************************************************)
(* Types *)
(*****************************************************************************)

⟨type Hexsha.t 33a⟩

⟨function Hexsha.is_hexsha 33c⟩

(*****************************************************************************)
(* Entry point *)
(*****************************************************************************)
```

```
(* start of copy-pasted code from ocaml-hex *)

let _hexa = "0123456789abcdef"
⟨constant Hexsha.hexa1 34f⟩
⟨constant Hexsha.hexa2 34g⟩

⟨function Hexsha.to_char 34a⟩

(*****************************************************************************)
(* Entry point *)
(*****************************************************************************)

⟨function Hexsha.of_sha 34d⟩

⟨function Hexsha.to_sha 33e⟩

⟨function Hexsha.read 44c⟩
⟨function Hexsha.write 52b⟩
```

E.1.58 version_control/index.mli

⟨version_control/index.mli 173a⟩≡

```
⟨type Index.stat_info 39d⟩
⟨type Index.mode 39e⟩
⟨type Index.time 40a⟩

⟨type Index.entry 39c⟩

⟨type Index.t 39b⟩

⟨signature Index.empty 40c⟩
⟨signature Index.mk_entry 40e⟩

⟨signature Index.stat_info_of_lstats 40g⟩
⟨signature Index.mode_of_perm 93b⟩
⟨signature Index.perm_of_mode 69c⟩

⟨signature Index.read 47b⟩
⟨signature Index.write 53c⟩

⟨signature Index.remove_entry 64d⟩
⟨signature Index.add_entry 63e⟩

⟨signature Index.trees_of_index 67c⟩
```

E.1.59 version_control/index.ml

⟨version_control/index.ml 173b⟩≡

```
⟨copyright ocaml-git 15c⟩
open Common
open Ord.Operators
open Fpath_.Operators

(*****************************************************************************)
(* Prelude *)
(*****************************************************************************)
```

```
(*
 * Most of the code below derives from: https://github.com/mirage/ocaml-git
 * in index.ml and git_unix.ml
 *)

(*****************************************************************************)
(* Types *)
(*****************************************************************************)
```

⟨*type* Index.stat_info 39d⟩
⟨*type* Index.mode 39e⟩
⟨*type* Index.time 40a⟩

⟨*type* Index.entry 39c⟩

```
(* less: extensions *)
```

⟨*type* Index.t 39b⟩

⟨*constant* Index.empty 40d⟩

```
(*****************************************************************************)
(* Helpers *)
(*****************************************************************************)

(*
(* Index entries are sorted by the byte sequence that comprises the
   entry name; with a secondary comparison of the stage bits from the
   <ENTRY_FLAGS> if the entry name byte sequences are identical. *)
let compare_entries e1 e2 =
  match String.compare e1.name e2.name with
  | 0 -> Hash.Blob.compare e2.id e1.id
  | i -> i
*)
```

⟨*function* Index.stat_info_of_lstats 40h⟩

⟨*function* Index.mk_entry 40f⟩

⟨*function* Index.perm_of_mode 69d⟩

⟨*function* Index.mode_of_perm 93c⟩

```
(*****************************************************************************)
(* Add/Del *)
(*****************************************************************************)
```

⟨*function* Index.remove_entry 65⟩

⟨*function* Index.add_entry 63f⟩

```
(*****************************************************************************)
(* tree of index *)
(*****************************************************************************)
```

⟨*type* Index.dir 68c⟩
⟨*type* Index.dir_entry 68d⟩
⟨*type* Index.dirs 68b⟩

```
(* the code in this section derives from dulwich *)
```

⟨*function* Index.add_dir 68f⟩

⟨*function* Index.build_trees 69b⟩

⟨*function* Index.trees_of_index 68a⟩

```
(*****************************************************************************)
(* index of tree *)
(*****************************************************************************)
(* See repository.set_worktree_and_index_to_tree() *)

(*****************************************************************************)
(* IO *)
(*****************************************************************************)

(* was in IO.BigEndian and IO.ml before but better to moved here
 * so we can compile most of xix
 *)
let read_real_i32 ch =
  let big = Int32.shift_left (Int32.of_int (IO.read_byte ch)) 24 in
  let ch3 = IO.read_byte ch in
  let ch2 = IO.read_byte ch in
  let ch1 = IO.read_byte ch in
  let base = Int32.of_int (ch1 lor (ch2 lsl 8) lor (ch3 lsl 16)) in
  Int32.logor base big

let write_real_i32 ch n =
  let base = Int32.to_int n in
  let big = Int32.to_int (Int32.shift_right_logical n 24) in
  IO.write_byte ch big;
  IO.write_byte ch (base lsr 16);
  IO.write_byte ch (base lsr 8);
  IO.write_byte ch base
```

⟨*function* Index.read_time 48b⟩

⟨*function* Index.write_time 54b⟩

⟨*function* Index.read_mode 48c⟩

⟨*function* Index.write_mode 54a⟩

⟨*function* Index.read_stat_info 48a⟩

⟨*function* Index.write_stat_info 53f⟩

⟨*function* Index.read_entry 47e⟩

⟨*function* Index.write_entry 53e⟩

⟨*function* Index.read_entries 47d⟩

⟨*function* Index.read 47c⟩

⟨*function* `Index.write` 53d⟩

Uses `Common.spf()`, `IO.BigEndian.read_real_i32()`, `IO.BigEndian.read_ui16()`, `IO.BigEndian.write_i32()`, `IO.BigEndian.write_real_i32()`, `IO.really_nread()`, `Index.dir_entry`, `Index.mode.Gitlink` 132b, `Index.mode.Link` 39e, `Index.read_entries()` 173b, `Index.stat_info.uid` 39d, `Index.time.lsb32` 40a, and `Index.time.nsec` 40a.

E.1.60 version_control/main.ml

⟨`version_control/main.ml` 176⟩≡

```
⟨copyright ocamlgit 15a⟩
open Common

(*****************************************************************************)
(* Prelude *)
(*****************************************************************************)
(* An OCaml port of git, a distributed version control system.
 *
 * Some of the code derives from dulwich (a clone of git in Python)
 * and ocaml-git (another clone of git in OCaml).
 *
 * Main limitations compared to git/dulwich/ocaml-git:
 *  -??
 *
 * todo:
 *  - a simplified version where just Marshall data instead of using
 *    the specific git format. Save many LOC?
 *  - ??
 *
 * The code of ocamlgit uses code from
 *  - ocaml-git
 *  - ocaml-hex
 *  - camlzip
 *  - extlib
 *  - uuidm
 *
 * ocamlgit uses code from ocaml-git. However, ocamlgit is simpler because
 * it does not use fancy features of OCaml or fancy libraries:
 *  - no functor, include, module types, polymorphic variants, keyword args,
 *    or excessive nested modules. KISS.
 *  - no functorized Set and Map so no need for hash(), compare(), and equal()
 *    boilerplate functions everywhere
 *  - hardcoded use of SHA1, so no need functors taking Git.DIGEST and HashIO
 *  - no support for Mirage, so no need to parametrize many things,
 *    no need Fs module, no need lwt
 *  - no disk vs mem, just disk, so again need less functors
 *  - hardcoded use of zlib so no need functors taking inflate signature
 *  - no support for filename requiring special escapes
 *  - no use of Cstruct or Mstruct or Bigarray (simply use IO.ml and Bytes)
 *  - no logs
 *  - no fmt (use ocamldebug or ocamltarzan dumpers)
 *  - no sexplib
 *
 * good stuff I took from ocaml-git:
 *  - dotgit (more readable than commondir in dulwich)
 *  - '/' operator (more readable than all those os.path.join in dulwich)
 *  - Hash.Tree.t, Hash.Commit.t, Hash.Blob.t more precise hash types
 *    (but they are not statically checked in ocamlgit)
 *  - TODO GRI (generalization of URI)
```

```
 * good stuff I wish I could take from dulwich:
 *  - hashtbl [] overloading, so can do r.refs["refs/tags/"+tag] = obj.id
 *    (thx to __setitem__ and __getitem__, but true that it also entails
 *     lots of boilerplate code)
 *
 * TODO:
 *  - look go-git, better basis? more complete than ocamlgit?
 *)

(*****************************************************************************)
(* Helpers *)
(*****************************************************************************)

⟨constant Main.commands 56c⟩

⟨function Main.usage 57a⟩

(*****************************************************************************)
(* Entry point *)
(*****************************************************************************)

⟨function Main.main 56b⟩

⟨toplevel Main._1 56a⟩
```

Uses Cmds.main_commands 55b, Common.spf(), and version_control/Main.main() 176.

E.1.61 version_control/merge.ml

⟨version_control/merge.ml 177a⟩≡

```
(*
https://github.com/schuhschuh/cmake-basis/blob/master/src/utilities/python/diff3.py
*)
```

E.1.62 version_control/objects.mli

⟨version_control/objects.mli 177b⟩≡

```
⟨type Objects.t 35d⟩

⟨signature Objects.read 41c⟩

⟨signature Objects.write 49d⟩
```

E.1.63 version_control/objects.ml

⟨version_control/objects.ml 177c⟩≡

```
⟨copyright ocamlgit 15a⟩
open Common

(*****************************************************************************)
(* Prelude *)
(*****************************************************************************)

(*****************************************************************************)
(* Types *)
(*****************************************************************************)
```

⟨*type* Objects.t 35d⟩

```
(*****************************************************************************)
(* IO *)
(*****************************************************************************)
```

⟨*function* Objects.read 41d⟩

⟨*function* Objects.write 50a⟩

E.1.64 version_control/refs.mli

⟨version_control/refs.mli 178a⟩≡

⟨*type* Refs.refname 37d⟩

⟨*type* Refs.t 37g⟩

⟨*type* Refs.ref_content 38b⟩

⟨*signature* Refs.default_head_content 38c⟩

⟨*signature* Refs.is_valid_refname 37e⟩

⟨*signature* Refs.read 45f⟩
⟨*signature* Refs.write 52h⟩

⟨*signature* Refs.string_of_ref 37h⟩

E.1.65 version_control/refs.ml

⟨version_control/refs.ml 178b⟩≡

```
open Common
open Regexp_.Operators

(*****************************************************************************)
(* Types *)
(*****************************************************************************)
```

⟨*type* Refs.refname 37d⟩

⟨*type* Refs.t 37g⟩

⟨*type* Refs.ref_content 38b⟩

⟨*constant* Refs.default_head_content 38d⟩

⟨*function* Refs.is_valid_refname 37f⟩

```
(*****************************************************************************)
(* Dumper *)
(*****************************************************************************)
```

⟨*function* Refs.string_of_ref 38a⟩

```
(*****************************************************************************)
(* IO *)
(*****************************************************************************)
```

⟨*function* `Refs.read` 45g⟩

⟨*function* `Refs.write` 52i⟩

E.1.66 version_control/repository.mli

⟨`version_control/repository.mli` 179⟩≡

⟨*type* `Repository.t` 35a⟩

⟨*type* `Repository.objectish` 38f⟩

```
(* repo *)
```
⟨*signature* `Repository.init` 59b⟩
⟨*signature* `Repository.open_` 61a⟩
⟨*signature* `Repository.find_dotgit_root_and_open` 61d⟩

```
(* objects *)
```
⟨*signature* `Repository.read_obj` 41a⟩
⟨*signature* `Repository.read_objectish` 46e⟩
⟨*signature* `Repository.read_commit` 45b⟩
⟨*signature* `Repository.read_tree` 43e⟩
⟨*signature* `Repository.read_blob` 42f⟩
⟨*signature* `Repository.add_obj` 49a⟩
⟨*signature* `Repository.has_obj` 102a⟩

```
(* refs *)
```
⟨*signature* `Repository.read_ref` 45d⟩
⟨*signature* `Repository.write_ref` 52f⟩
⟨*signature* `Repository.follow_ref` 46a⟩
⟨*signature* `Repository.follow_ref_some` 46c⟩
⟨*signature* `Repository.all_refs` 74a⟩
⟨*signature* `Repository.set_ref` 100b⟩
⟨*signature* `Repository.del_ref` 75j⟩
```
(* atomic op *)
```
⟨*signature* `Repository.add_ref_if_new` 71c⟩
⟨*signature* `Repository.set_ref_if_same_old` 71e⟩

```
(* index *)
```
⟨*signature* `Repository.read_index` 46g⟩
⟨*signature* `Repository.write_index` 53a⟩
⟨*signature* `Repository.add_in_index` 63a⟩

⟨*signature* `Repository.commit_index` 67a⟩
⟨*signature* `Repository.set_worktree_and_index_to_tree` 77c⟩

```
(* packs *)
```

```
(* misc *)
```
⟨*signature* `Repository.walk_dir` 145a⟩

```
val parse_objectish: string -> objectish
```

E.1.67 version_control/repository.ml

⟨version_control/repository.ml 180⟩≡
```
⟨copyright ocamlgit 15a⟩
open Common
open Fpath_.Operators

(*****************************************************************************)
(* Prelude *)
(*****************************************************************************)
(* API to access repository data (objects, index, refs, packs).
 *
 * less: use nested modules for objects, index, refs below?
 *)

(*****************************************************************************)
(* Types *)
(*****************************************************************************)

⟨type Repository.t 35a⟩

⟨constant Repository.SlashOperator 35c⟩

⟨constant Repository.dirperm 60a⟩

⟨type Repository.objectish 38f⟩

(*****************************************************************************)
(* Helpers *)
(*****************************************************************************)

⟨function Repository.hexsha_to_filename 35b⟩

⟨function Repository.ref_to_filename 38e⟩

⟨function Repository.index_to_filename 39a⟩

⟨function Repository.with_file_out_with_lock 50d⟩


(* move in common.ml? *)
⟨function Repository.with_opendir 146a⟩

(* move in common.ml? (but remove .git specific stuff) *)
⟨function Repository.walk_dir 145b⟩

(*****************************************************************************)
(* Refs *)
(*****************************************************************************)

⟨function Repository.read_ref 45e⟩

⟨function Repository.follow_ref 46b⟩

⟨function Repository.follow_ref_some 46d⟩

⟨function Repository.add_ref_if_new 71d⟩

⟨function Repository.del_ref 76a⟩
```

⟨*function* Repository.set_ref_if_same_old 72⟩

⟨*function* Repository.set_ref 100c⟩

⟨*function* Repository.write_ref 52g⟩

⟨*function* Repository.all_refs 74b⟩

```
(*****************************************************************************)
(* Objects *)
(*****************************************************************************)
```

⟨*function* Repository.read_obj 41b⟩

⟨*function* Repository.read_commit 45c⟩
⟨*function* Repository.read_tree 43f⟩
⟨*function* Repository.read_blob 42g⟩

⟨*function* Repository.read_objectish 46f⟩

⟨*function* Repository.add_obj 49b⟩

⟨*function* Repository.has_obj 102b⟩

```
(*****************************************************************************)
(* Index *)
(*****************************************************************************)
```

⟨*function* Repository.read_index 47a⟩

⟨*function* Repository.write_index 53b⟩

⟨*function* Repository.content_from_path_and_unix_stat 64a⟩

⟨*function* Repository.add_in_index 63b⟩

```
(*****************************************************************************)
(* Commit *)
(*****************************************************************************)

(* less: move to cmd_commit.ml? *)
```
⟨*function* Repository.commit_index 67b⟩

```
(*****************************************************************************)
(* Checkout and reset *)
(*****************************************************************************)
```

⟨*function* Repository.build_file_from_blob 78c⟩

⟨*function* Repository.set_worktree_and_index_to_tree 77d⟩

```
(*****************************************************************************)
(* Packs *)
(*****************************************************************************)

(*****************************************************************************)
(* Repo init/open *)
```

```
(*****************************************************************************)
```

⟨*function* `Repository.init` 59c⟩

⟨*function* `Repository.open_` 61b⟩

⟨*function* `Repository.find_dotgit_root_and_open` 61e⟩

```
let parse_objectish _str =
  raise Todo
```

E.1.68 version_control/sha1.mli

⟨`version_control/sha1.mli` 182a⟩≡

⟨*type* `Sha1.t` 32a⟩

⟨*signature* `Sha1.sha1` 32b⟩

⟨*signature* `Sha1.read` 43b⟩
⟨*signature* `Sha1.write` 51e⟩

⟨*signature* `Sha1.is_sha` 32c⟩

E.1.69 version_control/sha1.ml

⟨`version_control/sha1.ml` 182b⟩≡
⟨*copyright uuidm* **??**⟩

```
(*****************************************************************************)
(* Prelude *)
(*****************************************************************************)
(*
 * Most of the code below comes from: https://github.com/dbuenzli/uuidm
 * (this code is also copy-pasted and used in git-mirage)
 *
 * alternatives:
 *  - https://github.com/vincenthz/ocaml-sha
 *    implements algorithm in highly optimized C code and then provides
 *    bindings to call this C code
 *  - https://github.com/xavierleroy/cryptokit/
 *    by Xavier Leroy, but also uses C code
 *  - nocrypto
 *    seems like the official crypto lib, but also uses C code
 *  - md5sum, produces 128-bit hash value (sha1 is 20 bytes so 160 bits)
 *)

(*****************************************************************************)
(* Types *)
(*****************************************************************************)
```

⟨*type* `Sha1.t` 32a⟩

⟨*function* `Sha1.is_sha` 32d⟩

```
(*****************************************************************************)
(* Entry points *)
```

```
(*****************************************************************************)
```

⟨*function* `Sha1.read` 43c⟩

⟨*function* `Sha1.write` 51f⟩

```
(* start of copy-pasted code from uuidm *)
```

⟨*function* `Sha1.sha1` 110b⟩

E.1.70 version_control/tree.mli

⟨`version_control/tree.mli` 183a⟩≡

⟨*type* `Tree.perm` 36d⟩

⟨*type* `Tree.entry` 36b⟩

⟨*type* `Tree.t` 36a⟩

⟨*type* `Tree.hash` 36c⟩

⟨*signature* `Tree.read` 42i⟩
⟨*signature* `Tree.write` 51b⟩

⟨*signature* `Tree.show` 83b⟩

⟨*signature* `Tree.walk_tree` 78a⟩

⟨*signature* `Tree.walk_trees` 92a⟩

E.1.71 version_control/tree.ml

⟨`version_control/tree.ml` 183b⟩≡

⟨*copyright ocaml-git* 15c⟩

```
open Common
open Ord.Operators
open Fpath_.Operators

(*****************************************************************************)
(* Prelude *)
(*****************************************************************************)
(*
 *
 * Most of the code below derives from: https://github.com/mirage/ocaml-git
 *)

(*****************************************************************************)
(* Types *)
(*****************************************************************************)
```

⟨*type* `Tree.perm` 36d⟩

⟨*type* `Tree.entry` 36b⟩

⟨*type* `Tree.t` 36a⟩

⟨*type* Tree.hash 36c⟩

```
(*****************************************************************************)
(* Walk *)
(*****************************************************************************)
```

⟨*function* Tree.walk_tree 78b⟩

⟨*function* Tree.walk_trees 92b⟩

```
(*****************************************************************************)
(* IO *)
(*****************************************************************************)
```

⟨*function* Tree.perm_of_string 43d⟩

⟨*function* Tree.string_of_perm 51g⟩

⟨*function* Tree.read_entry 43a⟩

⟨*function* Tree.write_entry 51d⟩

⟨*function* Tree.read 42j⟩

⟨*function* Tree.write 51c⟩

```
(*****************************************************************************)
(* Show *)
(*****************************************************************************)
```

⟨*function* Tree.show 83c⟩

E.1.72 version_control/unzip.mli

⟨version_control/unzip.mli 184⟩≡

```
(*
 * Unzip - inflate format decompression algorithm
 * Copyright (C) 2004 Nicolas Cannasse
 * Compliant with RFC 1950 and 1951
 *
 * This library is free software; you can redistribute it and/or
 * modify it under the terms of the GNU Lesser General Public
 * License as published by the Free Software Foundation; either
 * version 2.1 of the License, or (at your option) any later version,
 * with the special exception on linking described in file LICENSE.
 *
 * This library is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
 * Lesser General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public
 * License along with this library; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
 *)
(** Decompression algorithm.
```

```
  Unzip decompression algorithm is compliant with RFC 1950 and 1951 which
  are describing the "inflate" algorithm used in most popular file formats.
  This format is also the one used by the popular ZLib library.
*)
```

⟨*type* `Unzip.error_msg` 114b⟩

⟨*exception* `Unzip.Error` 114c⟩

⟨*signature* `Unzip.inflate` 112b⟩

```
(* internals *)
type t
```

⟨*signature* `Unzip.inflate_init` 117d⟩
⟨*signature* `Unzip.inflate_data` 118a⟩

```
val debug: bool ref
```

E.1.73 version_control/unzip.ml

⟨`version_control/unzip.ml` 185⟩≡
⟨*copyright ocaml-unzip* **??**⟩

```
(**************************************************************************)
(* Types *)
(**************************************************************************)

let debug = ref false
```

⟨*type* `Unzip.huffman` 112c⟩

⟨*type* `Unzip.adler32` 121i⟩

⟨*type* `Unzip.window` 113a⟩

⟨*type* `Unzip.state` 113e⟩

⟨*type* `Unzip.t` 113f⟩

⟨*type* `Unzip.error_msg` 114b⟩

⟨*exception* `Unzip.Error` 114c⟩

⟨*function* `Unzip.error` 114d⟩

```
(* ************************************************************************ *)
(* HUFFMAN TREES *)
```

⟨*function* `Unzip.tree_depth` 125b⟩

⟨*function* `Unzip.tree_compress` 125a⟩

⟨*function* `Unzip.make_huffman` 115e⟩

```
(* ************************************************************************ *)
(* ADLER32 (CRC) *)
```

⟨*function* `Unzip.adler32_create` 122c⟩

⟨*function* `Unzip.adler32_update` 122e⟩

⟨*function* `Unzip.adler32_read` 122h⟩

```
(* ************************************************************************ *)
(* WINDOW *)
```

⟨*constant* `Unzip.window_size` 113b⟩
⟨*constant* `Unzip.buffer_size` 113c⟩

⟨*function* `Unzip.window_create` 113d⟩

⟨*function* `Unzip.window_slide` 117a⟩

⟨*function* `Unzip.window_add_bytes` 116e⟩

⟨*function* `Unzip.window_add_char` 116g⟩

⟨*function* `Unzip.window_get_last_char` 126a⟩

⟨*function* `Unzip.window_available` 117b⟩

⟨*function* `Unzip.window_checksum` 122g⟩

```
(* ************************************************************************ *)
```

⟨*constant* `Unzip.len_extra_bits_tbl` 120e⟩
⟨*constant* `Unzip.len_base_val_tbl` 120f⟩
⟨*constant* `Unzip.dist_extra_bits_tbl` 121a⟩
⟨*constant* `Unzip.dist_base_val_tbl` 121b⟩
⟨*constant* `Unzip.code_lengths_pos` 124c⟩

⟨*constant* `Unzip.fixed_huffman` 115d⟩

⟨*function* `Unzip.get_bits` 114h⟩

⟨*function* `Unzip.get_bit` 114g⟩

⟨*function* `Unzip.get_rev_bits` 120h⟩

⟨*function* `Unzip.reset_bits` 114f⟩

⟨*function* `Unzip.add_bytes` 115a⟩

⟨*function* `Unzip.add_char` 115b⟩

⟨*function* `Unzip.add_dist_one` 125e⟩

⟨*function* `Unzip.add_dist` 121e⟩

⟨*function* `Unzip.apply_huffman` 115c⟩

⟨*function* `Unzip.inflate_lengths` 124d⟩

⟨*function* `Unzip.inflate_loop` 118d⟩

⟨*function* `Unzip.inflate_data` 118b⟩

⟨*function* `Unzip.inflate_init` 117e⟩

```
(*****************************************************************************)
(* Entry point *)
(*****************************************************************************)
```

⟨*function* `Unzip.inflate` 117c⟩

Uses `IO.create_in()`, `IO.read_byte()`, `Unzip.adler32.a1` 121i, `Unzip.adler32.a2` 121i, `Unzip.adler32_create()`, `Unzip.apply_huffman()`, `Unzip.buffer_size` 185, `Unzip.get_bits()` 115d, `Unzip.get_rev_bits()` 114g, `Unzip.huffman.NeedBits` 124e, `Unzip.inflate_init()` 118b, `Unzip.t.zfinal` 113f, `Unzip.t.zhuffman` 113f, `Unzip.t.zinput` 113f, `Unzip.t.zoutpos` 113f, `Unzip.t.zoutput` 113f, and `Unzip.window.wcrc`.

E.1.74 version_control/user.mli

⟨`version_control/user.mli` 187a⟩≡

⟨*type* `User.tz_offset` 37b⟩

⟨*type* `User.t` 37a⟩

⟨*signature* `User.read` 44d⟩
⟨*signature* `User.write` 52c⟩

⟨*signature* `User.string_of_date` 84a⟩

E.1.75 version_control/user.ml

⟨`version_control/user.ml` 187b⟩≡
⟨*copyright ocaml-git* 15c⟩
`open Common`

```
(*****************************************************************************)
(* Prelude *)
(*****************************************************************************)
(*
 *
 * Most of the code below derives from: https://github.com/mirage/ocaml-git
 *)

(*****************************************************************************)
(* Types *)
(*****************************************************************************)
```

⟨*type* `User.tz_offset` 37b⟩

⟨*type* `User.t` 37a⟩

```
(* less: default_tz_offset ? *)

(*****************************************************************************)
(* IO *)
(*****************************************************************************)
```

⟨*function* `User.sign_of_char` 45a⟩

⟨*function* `User.char_of_sign` 84c⟩

⟨*function* `User.read` 44e⟩

⟨*function* `User.write_date` 52e⟩

⟨*function* `User.write` 52d⟩

```
(*****************************************************************************)
(* Show *)
(*****************************************************************************)
```

⟨*function* `User.string_of_date` 84b⟩

E.1.76 version_control/zip.mli

⟨`version_control/zip.mli` 188a⟩≡

E.1.77 version_control/zip.ml

⟨`version_control/zip.ml` 188b⟩≡

E.1.78 version_control/zlib.mli

⟨`version_control/zlib.mli` 188c⟩≡
⟨*copyright camlzip* **??**⟩

⟨*exception* `Zlib.Error` 126e⟩

⟨*signature* `Zlib.compress` 126c⟩

⟨*signature* `Zlib.compress_direct` 127b⟩

⟨*signature* `Zlib.uncompress` 127d⟩

```
type stream
```

⟨*type* `Zlib.flush_command` 126g⟩

```
external deflate_init: int -> bool -> stream = "camlzip_deflateInit"
external deflate:
  stream -> bytes -> int -> int -> bytes -> int -> int -> flush_command
         -> bool * int * int
  = "camlzip_deflate_bytecode" "camlzip_deflate"
external deflate_end: stream -> unit = "camlzip_deflateEnd"

external inflate_init: bool -> stream = "camlzip_inflateInit"
external inflate:
  stream -> bytes -> int -> int -> bytes -> int -> int -> flush_command
         -> bool * int * int
  = "camlzip_inflate_bytecode" "camlzip_inflate"
external inflate_end: stream -> unit = "camlzip_inflateEnd"

external update_crc: int32 -> bytes -> int -> int -> int32
                   = "camlzip_update_crc32"
external update_crc_string: int32 -> string -> int -> int -> int32
                   = "camlzip_update_crc32"
```

E.1.79 version_control/zlib.ml

⟨version_control/zlib.ml 189⟩≡
⟨*copyright camlzip* ??⟩

⟨*exception* Zlib.Error 126e⟩

⟨*toplevel* Zlib._1 126f⟩

```
type stream
```

⟨*type* Zlib.flush_command 126g⟩

```
external deflate_init: int -> bool -> stream = "camlzip_deflateInit"
external deflate:
  stream -> bytes -> int -> int -> bytes -> int -> int -> flush_command
         -> bool * int * int
  = "camlzip_deflate_bytecode" "camlzip_deflate"
external deflate_end: stream -> unit = "camlzip_deflateEnd"

external inflate_init: bool -> stream = "camlzip_inflateInit"
external inflate:
  stream -> bytes -> int -> int -> bytes -> int -> int -> flush_command
         -> bool * int * int
  = "camlzip_inflate_bytecode" "camlzip_inflate"
external inflate_end: stream -> unit = "camlzip_inflateEnd"

external update_crc: int32 -> bytes -> int -> int -> int32
                   = "camlzip_update_crc32"
external update_crc_string: int32 -> string -> int -> int -> int32
                   = "camlzip_update_crc32"
```

⟨*constant* Zlib.buffer_size 127a⟩

⟨*function* Zlib.compress 126d⟩

⟨*function* Zlib.compress_direct 127c⟩

⟨*function* Zlib.uncompress 127e⟩

Uses Zlib.flush_command 126g and Zlib.stream 189.

Glossary

VCS = Version Control System
DVCS = Distributed Version Control System
SHA1 = Secure Hash Algorithm 1
URL = Uniform Resource Locator

Indexes

Here is a list of the identifiers used, and where they appear. Underlined entries indicate the place of definition. This index is generated automatically.

Bibliography

[CS14] Scott Chacon and Ben Straub. *Pro Git*. 2014. Available at https://git-scm.com/book/en/v2. cited page(s) 13, 14

[EJ01] D. Eastlake and P. Jones. US Secure Hash Algorithm 1 (SHA1), September 2001. Available at http://www.ietf.org/rfc/rfc3174.txt. cited page(s) 19

[Gru86] Dick Grune. Concurrent versions system, a method for independent cooperation. Technical report, Vrije Universiteit, Amsterdam, 1986. Unpublished but available at https://dickgrune.com/Books/Publications/Concurrent_Versions_System,_a_method_for_independent_cooperation.ps. cited page(s) 12

[Har08] Jon Harrop. *F# for Scientists*. Wiley, 2008. cited page(s) 14

[Knu92] Donald E. Knuth. *Literate Programming*. Center for the Study of Language and Information, 1992. For an introduction see http://en.wikipedia.org/wiki/Literate_Program. cited page(s) 15

[LDF+16] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The OCaml system (release 4.04): Documentation and user's manual*. November 2016. Available at http://caml.inria.fr/pub/docs/manual-ocaml/. cited page(s) 14

[O'S09] Brian O'Sullivan. *Mercurial: The Definitive Guide*. O'Reilly, 2009. Available at http://hgbook.red-bean.com/. cited page(s) 13

[Pad09] Yoann Padioleau. Syncweb, literate programming meets unison, 2009. https://github.com/aryx/syncweb. cited page(s) 15

[Pad14] Yoann Padioleau. *Principia Softwarica: The Plan 9 Kernel 9*. 2014. cited page(s) 19, 24

[Pau96] Lawrence C. Paulson. *ML for the Working Programmer*. University of Cambridge, 1996. Available at http://www.cl.cam.ac.uk/~lp15/MLbook/pub-details.html. cited page(s) 14

[Roc75] Marc J. Rochkind. The source code control system. *IEEE Transactions on Software Engineering*, pages 364–370, December 1975. Available at http://www.basepath.com/aup/talks/SCCS-Slideshow.pdf. cited page(s) 12

[sha93] Secure hash standard, April 1993. Latest version available at https://csrc.nist.gov/publications/detail/fips/180/4/final. cited page(s) 19

[Tic85] Walter F. Tichy. Rcs - a system for version control. *Software Practice and Experience*, pages 637–654, July 1985. Available at http://www.gnu.org/s/rcs/tichy-paper.pdf. cited page(s) 12

[WL99] Pierre Weis and Xavier Leroy. *Le Langage Caml*. Dunod, 1999. In french. Available at http://caml.inria.fr/pub/distrib/books/llc.pdf. cited page(s) 14