université PARIS-SACLAY | IUT D'ORSAY

Tarides

Département d'Informatique

Diplôme préparé : Informatique

Jury N° 5

Implémentation d'une librairie de caches vérifié en OCaml

Stage de fin de DUT Informatique 2022

Stage réalisé par **Charlène Gros**

Tuteur enseignant
Andrei Paskevich

Responsable en Entreprise
Clément Pascutto

Année universitaire 2021—2022
Stage réalisé du 19 Avril au 1er Juillet 2022

Rapport remis le 17 Juin 2022

Remerciements

Je souhaite tout d'abord remercier toute la société Tarides pour m'avoir accueillie si chaleureusement ainsi que pour l'atmosphère bienveillante qui règne dans les locaux. Ils ont toujours su être présents pour la bonne réalisation de mon stage. Je remercie particulièrement M. Clément Pascutto et M. Nicolas Goguey, qui ont cru en mes capacités pour mener à bien cette mission. Ils m'ont toujours épaulé et aidé sans hésiter, toujours dans la bonne humeur. Grâce à eux, j'ai énormément appris notamment sur le plan technique.

Je tiens également à remercier M. Andrei Paskevich de m'avoir donné l'opportunité de réaliser ce stage au sein de Tarides. Je le remercie pour m'avoir aidée et accompagnée durant le stage, ainsi que pour sa bienveillance et sa gentillesse.

Résumé

Ce stage clôture mes deux années universitaires à l'IUT d'Orsay. Je l'ai effectué dans l'entreprise Tarides à Paris, spécialisée dans d'édition de logiciels utilisant le langage de programmation OCaml.

Ma mission fut de fournir la bibliothèque `cachecache`, qui permet l'utilisation de plusieurs algorithmes d'éviction de cache. Un cache est une mémoire locale qui stocke des informations, rendant leur accès plus rapide que celui dans leur stockage externe. Il existe plusieurs algorithmes de remplacement dans le cache, suivant la valeur supprimée lorsque le cache est plein et qu'il faut en ajouter une nouvelle. J'ai commencé par produire des outils d'utilisation, ainsi qu'un module de statistiques permettant de comparer les futurs algorithmes. Par la suite, j'ai implémenté et assuré la vérification d'un algorithme d'éviction LFU (Least Frequently Used). Enfin, j'ai produit des tests ainsi que des benchmarks, permettant de comparer les temps d'exécution des stratégies dans plusieurs situations, pour connaître la plus optimale dans ces circonstances.

Table des matières

Première partie

Présentation de l'entreprise

1 Entreprise

Tarides est une entreprise d'édition de logiciels écrits dans le langage de programmation OCaml. Elle a été créé en 2018 à la suite d'un projet de recherche à l'Université de Cambridge par ses quatre fondateurs issus de l'Université de Cambridge (Royaume-Uni) et de l'IIT Madras (Inde). Ce projet donne également lieu à la création en parallèle par les mêmes fondateurs des entreprises OCaml Labs Consultancy (Royaume-Uni) et Segfault Systems (Inde). En 2022, OCaml Labs et Segfault System fusionnent avec Tarides et deviennent Tarides UK et Tarides India. L'objectif principal du groupe est de développer l'environnement OCaml pour le rendre plus accessible et développer son écosystème open-source ainsi que des projets industriels l'utilisant.

Les activités de Tarides s'organisent autour de cinq groupes :

— **Compilateur** : contribuer à l'évolution du compilateur OCaml. En particulier, assurer la maintenance et la stabilité des versions 4.x, améliorer les interactions avec l'outillage, participer au déploiement d'OCaml 5 et de la fonctionnalité multi-coeur.
— **Plateforme** : coordonner le développement de l'outillage et des ressources en ligne à destination des développeurs.
— **Écosystème** : participer à la production de bibliothèques de l'écosystème OCaml, par exemple Irmin et accompagner l'adoption d'OCaml 5.
— **Tests et Benchmarks** : inventer des outils de tests et de benchmarks (en particulier en intégration continue) à destination des mainteneurs de logiciels OCaml.
— **Services** : Accompagner des entreprises externes dans leur utilisation d'OCaml.

Localisation géographique. Tarides possède des locaux en France, en Inde et au Royaume-Uni et emploie des collaborateurs travaillant à distance, par exemple depuis les États-Unis, l'Australie, ou encore le Japon. Les locaux français sont localisés 21 rue Rollin, dans le 5ème arrondissement de Paris. Ils sont situés dans une rue très calme, tout proche du Panthéon et de la rue Mouffetard, rue assez passante surtout en période estivale. Cette dernière est garnie de restaurants divers et variés, du traiteur grec au burrito mexicain en passant par des ramens et de délicieux burgers. L'accessibilité en transport en commun n'est pas en reste, car la rue Rollin est à 7 minutes à pied de l'arrêt Luxembourg, sur la ligne du RER B.

Méthodologie open-source. La totalité des projets réalisés par Tarides suivent une méthodologie open-source, c'est-à-dire que les codes sources produits sont accessibles par tout le monde sur la plateforme GitHub, ainsi que les discussions relatives au développement. Les collaborateurs de l'entreprise, aussi bien des laboratoires de recherche comme le CNRS ou Inria, que des entreprises privées partagent les connaissances et les travaux dans le but de maximiser l'innovation, et le développement de nouveaux outils et technologies.

2 Équipe

J'ai intégré l'équipe Irmin, qui fait partie du groupe Écosystème. Cette équipe comporte cinq ingénieurs, dont trois basés à Paris. Ils sont chargés de développer Irmin, une bibliothèque de bases de données. Cette équipe existe depuis la création de Tarides et a pour principal client

la Fondation Tezos, qui supervise le développement de la cryptomonnaie Tezos ainsi que son écosystème. Des réunions avec les clients ont lieu chaque semaine, pour empêcher l'effet tunnel.

L'équipe organise également des réunions internes chaque semaine afin de discuter de ce qui a été réalisé et planifier les tâches à venir. Les sujets abordés sont notés dans un document partagé, pour que chaque collaborateur ajoute un point de discussion si besoin et puisse les consulter ultérieurement. Le reste des échanges se fait via la messagerie Slack. Enfin, comme certains membres de l'équipe ne sont pas francophones, le travail a lieu intégralement en anglais.

Deuxième partie
Missions

3 Contexte et objectifs

Ma mission dans ce stage s'inscrit dans l'environnement OCaml, qui est un langage majoritairement fonctionnel. Je travaille spécifiquement sur une bibliothèque de caches qui sera utilisée en particulier par Irmin afin d'en améliorer les performances.

3.1 OCaml

En programmation, plusieurs paradigmes existent et guident la manière d'approcher et de résoudre un problème. Ils orientent la façon de raisonner au niveau logique, mais aussi de l'implémentation, sans forcément être incompatibles entre eux.

OCaml est un langage multiparadigmes, c'est-à-dire que son utilisation concilie plusieurs approches de programmation. Ainsi, il obéit à la fois aux logiques de la programmation impérative, fonctionnelle et objet.

- Un programme impératif est organisé autour d'une succession d'opérations qui modifient l'état de la mémoire, en utilisant principalement des structures mutables.
- Dans la programmation fonctionnelle, la progression du calcul se fait par des applications de fonctions successives et le passage de valeurs immuables, plutôt que par la modification d'un état. Les fonctions elles-mêmes sont considérées comme des valeurs, il est donc possible de donner en paramètre d'une fonction une autre fonction.
- Enfin, la programmation objet considère les entités de programmation comme des objets ayant des attributs et des méthodes qui leur sont propres. Ce paradigme est très peu utilisé par les développeurs OCaml.

Modules. Un programme OCaml est structuré en modules. Un module est composé de deux éléments : l'interface (que l'on écrit dans des fichiers `.mli`) et l'implémentation (dans des fichiers `.ml`).

L'interface contient la signature du module, c'est-à-dire la liste des déclarations des types et fonctions (avec leurs types) qui sont exposés par le module. Par exemple, considérons un module de gestion d'utilisateurs ; il pourrait avoir la signature suivante. On peut noter que la définition du type `user` n'est pas exposée, on peut donc se servir des interfaces pour créer des barrières d'abstraction fortes.

```
1 type user
2 val create : name:string -> user
3 val creation_date : user -> float
```

L'implémentation, elle, contient les définitions du module et de ses éléments. Le module précédent pourrait ainsi être implémenté de cette manière :

```
type t = {
  name : string;
  id : int
  created_on : float;
}

let new_id =
  let c = ref 0 in
  fun () -> incr c; !c

let create ~name =
  { name; id = new_id (); created_on = Unix.time () }

let creation_date t = t.created_on
```

Le système de modules du langage permet également de créer des foncteurs, c'est-à-dire des applications de modules. En d'autres termes, ce sont des valeurs analogues aux fonctions, mais qui prennent en paramètre des modules et qui en rendent de nouveaux.

Enfin, les programmes OCaml sont multiplateformes. Le langage supporte la compilation vers le langage machine pour beaucoup d'architectures (par exemple ARM64 ou x86). Le langage s'adapte aussi à la compilation vers du bytecode, par nature très portable. Ce bytecode peut être lui-même compilé vers du JavaScript, avec la bibliothèque `js_of_ocaml`.

Ce langage se distingue de ceux que j'ai appris durant ma formation. Premièrement, je n'avais pratiquement connu que des langages orientés objet. Ensuite, c'est au niveau du typage du langage que j'ai ressenti une grande différence. Là où certains langages plus populaires sont permissifs et permettent parfois des changements de types pour une variable donnée ou des conversions implicites, OCaml possède une stricte gestion des types. En effet, le compilateur vérifie statiquement que les programmes sont bien typés et rapporte les erreurs à la compilation (avant l'exécution) ce qui rend le langage plus sûr et aussi facile à lire, car les types sont clairs et immuables. J'ai trouvé cependant que l'accompagnement de la recherche de bugs dans les programmes OCaml n'est pas un point fort du langage.

3.2 Irmin

Irmin est une bibliothèque open-source écrite en OCaml, permettant la construction de base de données. Contrairement à un système de bases de données classique, c'est un système clé-valeur, qui s'inspire de Git. Il est possible de créer des branches, faire diverger la base de données depuis une base parente, les fusionner et surtout de gérer les différentes versions. Cela laisse à l'utilisateur une très large liberté quant à l'utilisation de sa base de données.

On associe à chaque couple clé-valeur un nombre nommé `hash`. Le hash est un nombre signature calculé à partir des données qu'il contient. Ce nombre est assez grand pour que la probabilité de collision soit quasiment nulle, dans le but de considérer que chaque hash est unique. Cela forme une empreinte numérique unique de la donnée qui identifie son contenu. Un hash est ainsi associé aux données envoyées lorsque l'utilisateur pousse sa version locale du code en ligne. On appelle cela un *commit*. À chaque commit est associé un nouveau hash, ce qui permet de différencier les versions. Pour économiser la mémoire, on ne créé pas plusieurs fois la même donnée, donc les fichiers inchangés entre deux versions conservent le même hash. Il peut être utile de connaître les

anciennes versions d'une base de données, afin de contrôler la bonne gestion de cette dernière. C'est notamment le cas dans l'utilisation majeure de Irmin avec le groupe Tezos, dans lequel il est primordial de vérifier la véracité des ajouts dans la blockchain Tezos.

En plus de ces fonctionnalités, Irmin permet à l'utilisateur une grande liberté de personnalisation. En effet, s'il est possible de stocker de simples chaînes de caractère (comme dans Git), il est aussi possible de créer ses propres types de données stockés. L'utilisateur doit simplement fournir un type et une fonction de fusion qui seront utilisés pour gérer automatiquement les conflits lors de la fusion de branches.

```
type t
(** The type for user-defined contents. *)

val merge : t option Merge.t
(** The merge function over contents used to handle branch merges. *)
```

L'accès répété aux données d'Irmin peut nécessiter beaucoup de lectures sur le disque qui dégradent les performances, car ces lectures sont coûteuses en temps. L'objectif de ce stage est donc de contribuer à une bibliothèque de caches qui permettra d'accélérer les accès aux données.

3.3 Cachecache

Le cache est une mémoire temporaire qui fournit un accès plus rapide aux données en les stockant dans la RAM, plus rapide que le disque. Lors d'une tentative d'accès aux données, on se tourne vers le cache en priorité. Cependant, cette mémoire ne peut contenir l'entièreté des données, car la RAM est limitée. Dans ce cas, on va alors chercher sur le disque la donnée et l'ajouter au cache pour accélérer ses utilisations futures.

Lors de l'ajout d'une donnée, le cache peut être plein. Dans ce cas, il faut décider quelle donnée existante supprimer pour laisser place à la nouvelle. On implémente cette décision par des algorithmes de remplacement de cache. On peut par exemple choisir de supprimer la donnée la moins souvent accédée (LFU, *Least Frequently Used*), ou la moins récemment recherchée (LRU, *Least Recently Used*), en fonction de l'utilisation que l'on en a. Par exemple, la stratégie traquant la fréquence est plus intéressante si nous savons que les mêmes données vont être souvent réutilisées.

La bibliothèque de caches à laquelle je participe se nomme `cachecache` et est disponible à l'adresse `https://github.com/pascutto/cachecache`. À mon arrivée, `cachecache` implémentait l'algorithme de la `LRU`. C'est la stratégie *Least recently used* qui supprime, lorsque le cache est plein, la valeur la plus anciennement accédée. Elle est accompagnée d'un module de liste doublement chaînée spécifique à cette implémentation et de tests unitaires.

Cahier des charges. La bibliothèque `cachecache` va contenir plusieurs algorithmes de cache. Pour que cette bibliothèque soit facile d'utilisation, il faut que chaque algorithme fournit les mêmes fonctions d'accès aux données (de recherche, d'ajout, de suppression...). Il faut donc une unique interface pour tous les algorithmes (voir Fig. 1).

Il est également impératif de conserver la cohérence entre le cache et la base de donnée lors de son utilisation, c'est-à-dire qu'à chaque instant, tous les éléments présents dans le cache doivent nécessairement être présents dans la source (base de donnée...). En revanche, il est possible (et même normal) que certains éléments de la source ne soient pas présents dans le cache.

Chaque algorithme de cache doit être rapide et efficace en mémoire, afin de limiter son influence sur la durée et la consommation en mémoire de l'exécution du projet dans lequel il est utilisé. Enfin, chaque cache doit être borné en mémoire, c'est-à-dire avoir une capacité fixe donnée lors de la création du cache, qui permet de limiter la taille des données en mémoire.

```
module type Cache = sig
  type 'a t
  (** The type for caches holding values of type ['a]. *)

  type key
  (** The type for cache keys. *)

  val v : int -> 'a t
  (** [v cap] is a fresh cache with capacity [cap]. *)

  val size : 'a t -> int
  (** [size t] is the current number of bindings in [t]. *)

  val replace : 'a t -> key -> 'a -> unit
  (** [replace t k v] binds [k] to [v] in [t], replacing the
      existing binding for [k], if any. *)

  val mem : 'a t -> key -> bool
  (** [mem t k] is [true] iff [k] is bound in [t]. *)

  val find : 'a t -> key -> 'a
  (** [find t k] is the value bound to [k] in [t], if any.
      Raises [Not_found] otherwise. *)

  val remove : 'a t -> key -> unit
  (** [remove t k] removes the binding to [k] in [t]. *)
end
```

FIGURE 1 – Interface commune simplifiée des modules de cache

4 Déroulement

J'ai contribué à plusieurs aspects de la bibliothèque `cachecache`. Pour me familiariser avec le langage et les nouvelles structures, j'ai commencé par écrire un module de collecte de statistiques, dans le but de comparer les algorithmes de remplacement (Sec. 4.1). J'ai aussi implémenté une API alternative qui permet une utilisation différente du cache (Sec. 4.2) afin de me familiariser avec les foncteurs. J'ai ensuite conçu et implémenté un algorithme de remplacement qui élimine les données les moins fréquemment utilisées (Sec. 4.3), en utilisant deux versions de listes doublement chaînées (Sec. 4.4) que j'ai écrites. Enfin, j'ai réalisé des tests de performance dans le but de comparer les différents algorithmes et implémentations (Sec. 4.6). Deux jours étaient consacrés à un évènement nommé « `Hacking Days` », lors duquel tout le monde travaille sur un projet annexe de son choix, qu'il présentera le mois suivant. J'ai travaillé de mon côté sur l'implémentation de la fourmi de Langton (automate cellulaire) durant ces deux jours (Sec. 4.7).

Pour chaque module implémenté, correspond une `Pull Request` correspondant. Elle contient tous les `commits` de la création aux finitions du module.

4.1 Module de statistiques

L'ambition de `cachecache` étant de fournir plusieurs algorithmes de remplacement de cache, il est important de fournir un moyen permettant de les comparer afin que l'utilisateur puisse choisir la meilleure option. Pour cela, j'ai écrit un module collectant les statistiques d'accès aux données d'un cache, qui pourra être utilisé indifféremment par toutes les implémentations. Par exemple, on cherche à compter les requêtes réussies (*hits*), les requêtes échouées (*miss*), ou encore la taille maximale atteinte par le cache. Le type des statistiques est un simple type d'enregistrement contenant les différents compteurs. Le mot-clé `private` assure que l'utilisateur ne peut pas modifier ces compteurs, mais seulement les lire.

```
type t = private {
  mutable miss : int;  (** Number of misses. *)
  mutable hit : int;  (** Number of hits. *)
  mutable add : int;  (** Number of newly added bindings. *)
  mutable replace : int;  (** Number of overwritten bindings. *)
  mutable discard : int;  (** Number of discarded bindings. *)
  mutable remove : int;  (** Number of calls to [remove]. *)
  mutable clear : int;  (** Number of calls to [clear]. *)
  mutable max_size : int;  (** Maximum reached number of bindings. *)
}
```

On ajoutera ensuite une fonction permettant d'accéder en lecture à ces statistiques pour les utilisateurs du cache :

```
val stats : 'a t -> Stats.t
```

Il a donc fallu explorer et comprendre l'implémentation de `cachecache` existante, afin de dégager les différentes situations. Pour l'instrumentation des fonctions `mem` et `find`, il faut par exemple dissocier les appels qui réussissent à retourner une valeur (dans le cas ou la valeur existe dans le cache), c'est-à-dire les `hit`, des appels ne trouvant pas la valeur, les `miss`. J'ai aussi séparé les trois finalités possibles de l'appel de la fonction `Cache.replace` : soit la clé donnée en paramètre existe déjà, dans quel cas j'incrémente le compteur de `replace`, ou bien la clé n'existe pas. Dans ce dernier cas, le programme regarde l'état du cache : s'il est plein, alors la valeur ajoutée sera issue de la suppression d'une autre, ce qui incrémente le compteur `discard` sinon, la valeur sera simplement ajoutée et l'incrémentation du compteur `add` sera réalisée.

Pull request. `https://github.com/pascutto/cachecache/pull/5/`

4.2 Stratégies d'utilisation du cache

L'utilisation d'un cache fourni par `cachecache` par une application résulte souvent en la répétition d'un même motif de code : l'application cherche une donnée dans le cache puis, si elle ne la trouve pas, cherche dans la source (par exemple une base de donnée Irmin). Le même processus apparaît aussi pour les écritures et les suppressions, puisqu'il faut à tout moment maintenir la cohérence du cache.

```
1 try Cache.find cache key with Not_found -> Database.find cache key
```

Afin d'améliorer l'usage et de simplifier le code des applications, j'ai exploré différentes stratégies. Il existe deux principaux modèles d'utilisations du cache par une application (voir Fig. 2).

(a) Cache-aside. (b) Cache-through.

FIGURE 2 – Configuration d'utilisation d'un cache

Cache indirect *(Cache-Aside).* Dans cette configuration, l'application communique à la fois avec le cache et la base de donnée (voir Fig. 2a). Elle est responsable de gérer les exceptions levées par le cache, en particulier lorsqu'une clé n'est pas trouvée et qu'une requête dans la base de données est nécessaire.

Cache en ligne *(Cache-Through).* Ici, l'application communique uniquement avec le cache et c'est ce dernier qui est responsable des interactions avec la base de données (voir Fig. 2b). Cette configuration rend l'utilisation de la base données et du cache plus légère dans l'implémentation de l'application.

Dans `cachecache`, c'est la méthode indirecte qui était proposée à mon arrivée. J'y ai ajouté une nouvelle interface pour implémenter la stratégie en ligne et produit une suite de tests afin de m'assurer de sa correction.

Pour implémenter ce comportement, j'ai créé un foncteur `Through.Make`. Ce dernier prend en paramètre deux modules : une base de données (`DB`) et un cache (`Cache`). Il renvoie un nouveau module bases de données (`DB`) incluant un cache, c'est-à-dire que les requêtes passent automatiquement par le cache avant de consulter la base de données (voir Fig. 3).

Il existe deux catégories de requêtes dans la base de données : celles en lecture (comme `mem` ou `find`) et celles en écriture (comme `replace` ou `remove`). Dans le cas des requêtes en écriture, il faut porter une attention particulière pour conserver la cohérence du cache. Par exemple, il est

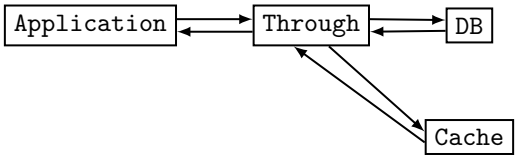

FIGURE 3 – Orchestration par le module `Through`

impératif de supprimer du cache une clé qui a été supprimée de la base de données, sans quoi le cache contiendrait des données obsolètes. En lecture, on effectue une première recherche dans le cache et une consultation de la base de données seulement lorsque la clé n'est pas présente dans le cache. Dans ce dernier cas, le cache est mis à jour avec le résultat de la requête.

Processus d'implémentation. J'ai commencé par créer de nouveaux types de modules, afin de modéliser les caches et les bases de données. D'un côté, le type de module `Cache` (Fig. 1) donne la signature de toutes les fonctions que doivent implémenter les modules de cache. De l'autre, le type de module base de données `DB` donne la signature des fonctions basiques d'accès, d'ajout et de suppression de valeurs que doit fournir une base de données :

```
module type DB = sig
  type t
  type key
  type value

  val v : unit -> t
  val mem : t -> key -> bool
  val find : t -> key -> value
  val replace : t -> key -> value -> unit
  val remove : t -> key -> unit
end
```

Enfin, j'ai écrit l'interface du module produit par le foncteur `Through.Make`. Il renvoie à la fois une base de données (augmentée d'un cache) et le cache lui-même dans un sous-module, afin de permettre à l'utilisateur de n'interroger que le cache s'il le souhaite. Cependant, ce dernier est légèrement modifié et n'implémente pas la fonction `replace`. C'est un choix nécessaire, dans le but de conserver la cohérence du cache. En effet, contrairement aux suppressions qui peuvent être effectuées uniquement sur le cache, l'utilisateur n'a pas le droit d'effectuer des ajouts et mise à jour de valeurs, si elles ne sont pas faites aussi sur la base de donnée associée.

```
module type Through = sig
  type t
  type key
  type value

  val v : unit -> t
  val mem : t -> key -> bool
  val find : t -> key -> value
  val replace : t -> key -> value -> unit
  val remove : t -> key -> unit

  module Cache : sig
    val size : t -> int
    val mem : t -> key -> bool
    val find : t -> key -> value
    val remove : t -> key -> unit
    val stats : t -> Stats.t
  end
end
```

Enfin, le foncteur prend deux arguments, un module de type `DB` et un module de type `Cache` et renvoie un module de type `Through`. Il faut bien veiller à contraindre les types `key` et `value` pour s'assurer qu'ils sont les mêmes dans tous les modules :

```
module Make (C : S.Cache) (DB : S.DB with type key = C.key) :
  S.Through with type key = DB.key and type value = DB.value
```

Pull request. `https://github.com/pascutto/cachecache/pull/6/`

4.3 Algorithme LFU *(Least Frequently Used)*

Une partie importante du stage a consisté en l'implémentation de l'algorithme d'éviction LFU. Cette stratégie consiste à remplacer la clé la moins utilisée lorsque le cache est plein, il faut donc traquer la fréquence d'accès aux clés. Dans un souci d'efficacité, `cachecache` promet une complexité en temps constant pour les algorithmes de caches fournis.

Complexité. Un algorithme peut utiliser différentes quantités de ressources, en temps ou en mémoire. Il est possible de mesurer un ordre de grandeur de ce temps ou de la place utilisée par l'algorithme et que l'on appelle la complexité algorithmique. Dans le meilleur des cas, nous avons une complexité en temps constant, notée `O(1)`. Cela signifie que le temps pris sera toujours le même s'il y a peu ou beaucoup de données. Dans le cas de la mise en cache, nous souhaitons minimiser les ressources utilisées aussi bien en temps qu'en mémoire.

Structures utilisées. Pour trouver un modèle de structures de données fournissant la complexité recherchée, j'ai analysé des structures de données existantes et sélectionné celles qui semblaient optimales, afin de fournir un accès rapide aux endroits où sont stockées les données importantes. Certaines structures de données sont connues pour avoir une complexité en temps constant. J'ai choisi d'utiliser des tables de hachage. C'est une structure de donnée où l'on stocke des associations clé-valeur, en utilisant une fonction de hachage. J'utilise aussi des listes doublement chaînées, où chaque élément possède un pointeur vers l'élément qui le précède et qui le succède.

Structure de la LFU. La structure permettant la LFU est constituée d'une table de hachage (`t.tbl`) qui contiendra les associations clé-valeur, d'une liste chaînée (`t.frequencies`) qui permettra de différencier les fréquences et enfin d'un ensemble de listes doublement chaînées (`t.lst`) qui conserveront, pour chaque fréquence, les clés présentes.

La liste (`t.frequencys`) est représentée verticalement en orange sur la figure 4) et contient dans chaque cellule la fréquence correspondante (un entier) et un pointeur la liste de `t.lists` contenant toutes les clés ayant cette fréquence (en vert et horizontalement). La table de hachage associe à chaque clé sa valeur, ainsi que deux pointeurs vers les cellules de `t.frequencies` et `t.lists` correspondantes.

Dans la figure 4, il existe onze valeurs dans le cache, donc autant de cellules vertes et d'entrées dans la table (omises sur le schéma). Elles sont réparties en quatre fréquences : quatre valeurs ont été accédées une fois, une valeur a été accédée trois fois etc. La valeur k_1 est associée à la valeur v_1 et a été accédée quatre fois et la valeur k_2 est associée à la valeur v_2 et a été accédée une seule fois.

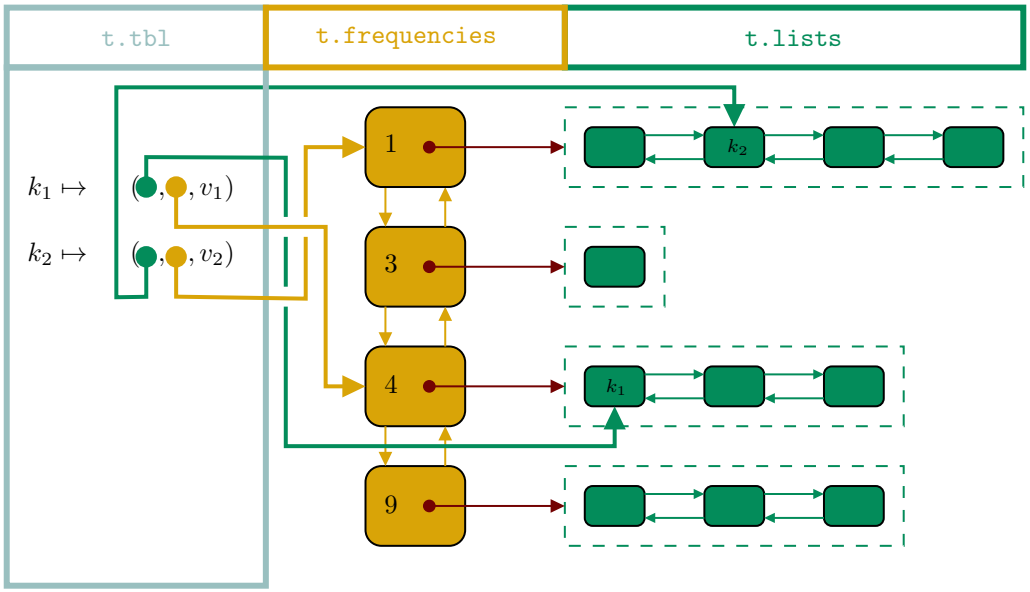

FIGURE 4 – Structure de la `LFU`

Types. Le type correspondant est décrit dans la figure 5. Ce type est bien sûr abstrait dans l'interface exposée à l'utilisateur.

Le nom du type `t` est arbitraire, mais par convention, on note `t` le nom du type principal du module. Il est polymorphe, c'est-à-dire qu'il est paramétré par un type `'a` qui pourra être instancié par n'importe quel type lors de son utilisation. Cela peut être un simple entier, une chaine de caractères, ou un type plus complexe défini par l'utilisateur. Il est nécessaire de mettre un type `'a` car nous ne connaissons pas au moment de la compilation la nature des données contenues dans le cache.

Ainsi, avec une clé `k` donnée, il est simple d'accéder directement à la valeur associée. Elle est le troisième et dernier élément du n-uplet associé à `k` dans la table de hachage `t.tbl`. Cette dernière fournit aussi toutes les informations pour trouver sa cellule dans les listes `t.lists`, ce qui est crucial puisqu'à chaque requête, il faut déplacer cette cellule vers la liste de fréquence plus élevée.

```
type freq_ptr = (int * key Dllist.list) Dllist.cell
(** Pointers to orange cells *)

type key_ptr = key Dllist.cell
(** Pointers to green cells *)

type 'a t = {
  lists : key Dllist.t
    (** A set of doubly linked lists containing keys (green) *);
  tbl : (freq_ptr * key_ptr * 'a) H.t
    (** A table associating keys with a triplet of two pointers and a value (teal) *);
  frequencies : (int * key Dllist.list) Dllist.list
    (** A doubly linked lists containing integers and pointers (orange) *);
  cap : int (** The capacity of the cache *);
  stats : Stats.t (** The statistics of the cache *);
}
```

FIGURE 5 – Type du cache LFU

Remplacement dans le cache. C'est la fonction `replace` qui marque souvent une différence entre les algorithmes d'éviction, car c'est lors d'un ajout dans le cache qu'il faut décider quelle donnée remplacer si le cache est plein.

Dans la stratégie `LFU`, c'est la donnée la moins fréquemment utilisée qui est supprimée du cache. J'ai allégé l'implémentation de `replace` (voir Fig. 6) en créant des fonctions intermédiaires `add` (qui gère les ajouts de nouvelles clés) et `update` (qui enregistre un accès en augmentant la fréquence associée à une clé), car elles sont utilisées plusieurs fois (ligne 5, 13 et 27 dans `replace`).

Sa structure est simple, j'effectue une recherche de la clé `k` donnée en entrée (ligne 1). Si cet appel ne lève aucune exception lors du `find` dans le cache ligne 3, cela signifie que la clé est déjà dans le cache. Dans ce cas, nous appelons `update` qui s'occupe de mettre à jour la fréquence de `k` dans `t.frequency`. Si la recherche lève l'exception `Not_Found` (rattrapée ligne 9), j'en conclus que `k` n'est pas dans le cache, il faut alors l'ajouter.

Dans ce cas, le cache peut ne pas être plein (vérifié avec la structure conditionnelle ligne 10). J'ajoute alors simplement avec `add`, qui ajoute l'association clé-valeur dans la table de hachage et dans la liste doublement chaînée `t.frequencies` avec la fréquence 1. Ce chaînon est créé s'il n'existe pas. Dans le cas où le cache est plein, il faut se débarrasser de la cellule avec la fréquence la plus basse, soit la tête de la liste de la tête de la liste doublement chaînée `t.frequencies` et supprimer la clé correspondante dans `t.tbl`. À ce moment-là, le cache n'est plus plein, il reste une place et nous ajoutons donc la donnée en appelant la fonction `add`.

Promesses. La `LFU` assure que les listes doublement chaînées ne sont jamais vides. Lorsque la liste d'une fréquence (en vert) devient vide, parce qu'une clé a été supprimée ou promue à la fréquence supérieure, la cellule de cette fréquence sera supprimée de `t.frequency`, car il n'a plus de raisons d'exister et de prendre de la place en mémoire inutile. Les pointeurs vers le prédécesseur et vers le successeur sont aussi modifiés pour conserver l'ordre de la liste doublement chaînée.

À chaque accès aux valeurs, il faut mettre à jour la table des fréquences, c'est-à-dire incrémenter la fréquence d'utilisation de la valeur accédée de 1. Grâce à l'implémentation actuelle, c'est facile à réaliser, il suffit d'accéder à la valeur lue, la retirer de sa liste actuelle et l'ajouter à la fin de la liste suivante. Cependant, il faut faire attention à étudier tous les cas. En effet, le successeur de

```
let replace t k v =
    try
      let _freq_index, _key_index, _value = H.find t.value k in
      (* replace *)
      let new_freq_index, new_key_index = update t k in
      Stats.replace t.stats;
      H.replace t.value k (new_freq_index, new_key_index, v)
    with Not_found ->
      Stats.add (H.length t.value + 1) t.stats;
      if H.length t.value < t.cap then
        (* add *)
        add t k v
      else
        (* discard *)
        let first_freq_index, _last_freq_index = Dllist.ends t.frequency in
        let _freq, freq_list = Dllist.get t.frequency first_freq_index in
        assert (not (Dllist.is_empty freq_list));
        let first_index, _last_index = Dllist.ends freq_list in
        let remove_key = Dllist.get freq_list first_index in
        Dllist.remove freq_list first_index;
        if Dllist.is_empty freq_list then
          Dllist.remove t.frequency first_freq_index;
        H.remove t.value remove_key;
        Stats.discard t.stats;
        add t k v
```

FIGURE 6 – Implémentation de la fonction `replace`

la fréquence actuelle n'est pas forcément de la bonne fréquence. Il faut donc créer un nouveau chaînon et l'insérer à la suite de la fréquence de la valeur. Après cela, il faut vérifier que la liste que la valeur quitte ne devienne pas vide, dans quel cas il faut supprimer ce chaînon.

Pull request. `https://github.com/pascutto/cachecache/pull/8`

4.4 Listes doublement chaînées

Une liste doublement chaînée est une structure de données dont chaque élément peut accéder grâce à des pointeurs à l'élément qui le précède et qui le succède. Cette structure est utilisée de nombreuses fois dans `cachecache`. Comme elle n'existe pas dans la bibliothèque standard de OCaml, j'ai décidé de l'implémenter. J'ai commencé par écrire une implémentation spécifique à l'usage par l'algorithme `LFU`, que l'on nommera ici version 1. Par la suite, j'ai trouvé certaines manières d'optimiser cette version, j'ai donc réalisé une nouvelle version qui convient aussi bien à la `LRU` qu'à la `LFU`.

4.4.1 Première version

Ce module fonctionne avec des cellules représentant chaque chaînon de la liste, que je représente dans un type `'a cell`. Chacune contient une valeur (`contents`) de type `'a` et pointe vers la

cellule la précédant (`prev`) et la succédant (`next`). J'ai veillé à ce que l'utilisateur ne puisse pas créér de cellules sans passer par le constructeur, en ajoutant le mot-clé `private` dans la définition du type `cell`. Sans cela, il existerait la possibilité de casser la cohérence de la liste, par exemple si les mauvais successeurs ou prédécesseurs sont ajoutés.

```
type 'a cell = private {
  mutable content : 'a;
  mutable prev : 'a cell;
  mutable next : 'a cell;
}
```

Dans une liste doublement chaînée, les valeurs des extrémités peuvent ne pas avoir de successeur (pour le dernier) ou de prédécesseur (pour le premier). Une solution naïve consiste à implémenter `prev` et `next` avec des `'a cell option` et mettre `None` lorsqu'ils sont absents. Cependant, cette approche introduit une indirection en mémoire (pour obtenir le contenu de l'option). Une meilleure solution consiste à faire pointer les extrémités vers elles-mêmes, puis de tester l'égalité physique (`==`) afin de différencier les cellules situées aux extrémités de la liste des autres internes. En OCaml, cela se traduit par un appel récursif lors de la création de la nouvelle cellule extérieure.

```
let rec new_cell = { content = value; prev = list.last; next = new_cell } in
```

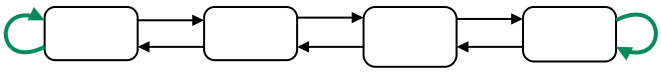

FIGURE 7 – Pointeurs récursifs

Cela peut être source d'erreurs si les opérateurs utilisés lors de tests ne sont pas les bons. J'ai découvert la différence entre les opérateurs `=` et `==`. Le langage permet de différencier les tests d'égalités dits structurels et physiques.

- L'égalité structurelle `"="`, retourne vrai si les deux objets comparés ont la même valeur. Lors de la comparaison de deux mêmes cellules cycliques, cela donne lieu à une boucle infinie.
- L'égalité physique `"=="`, retourne vrai si les deux objets comparés sont présents au même endroit en mémoire. J'ai donc dû utiliser cet opérateur pour comparer des cellules afin que le programme termine à-coup-sûr.

4.5 Faiblesses.

Si cette implémentation présente des performances tout à fait convenables (voir Sec. 4.6), je me suis proposée de les réimplémenter en utilisant des tableaux, cela permet de régler différentes faiblesses de la première version.

Améliorer la localité des données. Le CPU fait des appels à la mémoire pour obtenir les données sur lesquelles calculer. Même si la mémoire RAM est rapide, ces appels ont un coût. Pour cette raison, il stocke en interne des morceaux de la mémoire dans un cache. Si l'on souhaite micro-optimiser le programme, il faut faire le moins d'appels possible à la mémoire et utiliser au maximum les valeurs stockées dans le cache du CPU. L'implémentation des listes doublement chaînées avec des cellules a le désavantage d'éparpiller les cellules créées dans la mémoire. Ainsi, il y a peu de chances d'avoir toutes les cellules en un morceau de mémoire que le CPU pourra

stocker dans son cache. Une implémentation à base de tableaux permettrait de dé-fragmenter la mémoire, car dans un tableau toutes les données se suivent, ce qui peut économiser de nombreux appels à la mémoire (voir Fig. 8).

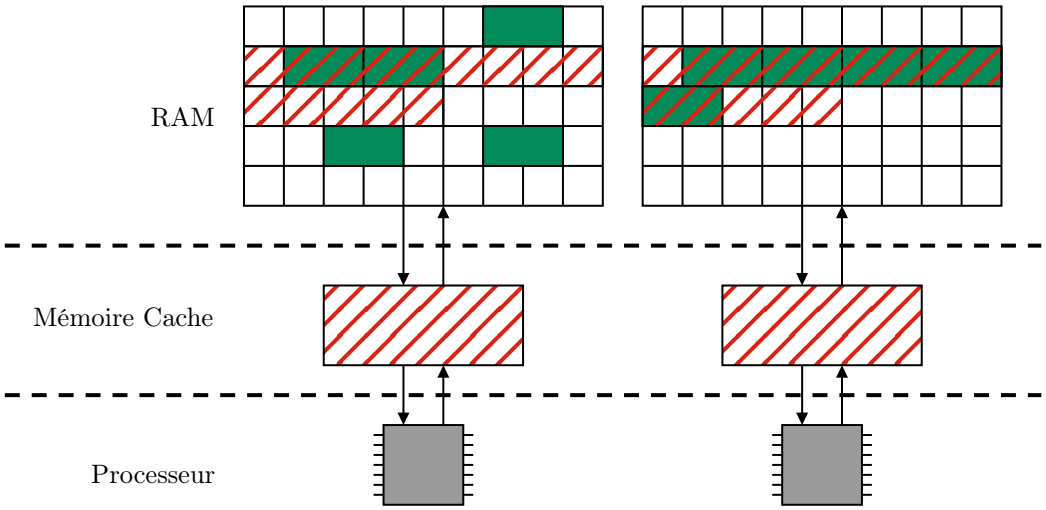

FIGURE 8 – Localité des données : les données fragmentées (à gauche) ont moins de chance de tenir dans le cache que les données compactes (à droite).

Limiter les allocations. Le ramasse-miettes (*Garbage Collector*, GC) est une fonctionnalité du langage permettant de gérer automatiquement les allocations mémoires. Régulièrement, le GC analyse la mémoire et supprime les objets non utilisés. Cette opération prend du temps et peut devenir importante lorsque le programme alloue beaucoup. Lors de l'accès aux données dans la version 1, on observe une grande quantité de petites allocations, puisqu'il faut créer de nouvelles cellules de listes à chaque requête sur le cache. Avec des tableaux, nous modifierions seulement des cellules dans des tableaux qui sont alloués une seule fois au début du programme, ce qui limitera le surplus d'allocations et donc d'appels au GC.

4.5.1 Version finale

J'ai repris l'implémentation de la liste doublement chaînée existante, utilisée dans la `LRU`, qui utilisait des tableaux afin d'encoder *une* liste. L'objectif est de n'utiliser qu'une implémentation pour toutes les stratégies. La `LRU` nécessite une liste doublement chaînée de taille fixe, mais la `LFU` en a besoin d'une pour les fréquences et chaque fréquence en possède une de taille variable. Un choix s'est posé afin de réadapter cette nouvelle stratégie : l'utilisation de vecteurs, ou bien la possibilité de créer une multitude de listes doublement chaînée pour un seul tableau de contenu.

Si l'on se donne comme objectif de ne pas allouer du tout, la première solution n'est pas bonne à terme car elle alloue encore lors du redimensionnement des tableaux et il est difficile à mettre en place lors de fonctions particulières.

Si l'on remarque que la liste des fréquences (orange) a une taille bornée par la capacité et que la somme des longueurs des listes des clés (vert) est aussi bornée par la capacité, on se rend compte que l'on peut stocker toutes ces listes dans des tableaux de taille `2 * cap`. La seconde solution s'est donc avérée la plus optimale : la `LRU` ne possède qu'un tableau, tandis que la `LFU`

pourra en implémenter plusieurs sur une même base. J'ai eu l'idée d'ajouter un tableau partagé entre toutes les cellules des listes doublement chaînées horizontales.

Le type `'a t` est partagé entre toutes les listes d'un même ensemble. Il possède une capacité maximale, c'est-à-dire la taille du cache dans le cas présent, ainsi que trois tableaux et un entier `t.free`.

```
type 'a t = {
  cap : int;
  witness : 'a;
  contents : 'a array;
  mutable free : int;
  prev : int array;
  next : int array;
}
```

Chaque tableau fournit des informations par rapport au même index. Par exemple, pour `i = 2`, il y a dans `t.contents.(i)` la valeur de la clé, dans `t.prev.(i)` l'index de son prédécesseur et dans `t.next.(i)`, l'index de son successeur. On peut facilement savoir si la valeur est à une extrémité de la liste doublement chaînée, car son successeur ou prédécesseur est égal à -1 dans ce cas là. Le type `'a list` en revanche représente chaque liste et conserve l'information du premier et du dernier élément de la liste, ainsi que sa taille et un pointeur vers la structure partagée.

```
type 'a list = {
  mutable first : int;
  mutable last : int;
  mutable size : int;
  t : 'a t;
}
```

Pour encadrer la création de liste doublement chaînée, on a donc besoin de deux constructeur, une pour initialiser le tableau commun et une autre pour créer une liste. La fonction `create` produit les tableaux qui devront être partagés et renvoie l'enregistrement (*record*) avec ces champs.

```
let create cap witness =
  {
    cap;
    witness;
    contents = Array.make cap witness;
    free = 0;
    prev = Array.init cap pred;
    next = Array.init cap (fun i -> if i = cap - 1 then -1 else succ i);
  }
```

Pour créer des listes dans ces tableaux partagés, il suffit d'utiliser `create_list` en donnant en paramètre le `t` créé précédemment.

```
let create_list t = { first = -1; last = -1; size = 0; t }
```

Le reste des opérations (par exemple `append`, `remove`) consiste à présent en des opérations sur les indices des tableaux.

Pull request. `https://github.com/pascutto/cachecache/pull/11`

4.5.2 Tests et spécification

Dans le but de tester la correction de la nouvelle implémentation, j'ai effectué des tests sur les algorithmes de cache qui étaient déjà vérifiés. J'ai d'abord utilisé les tests unitaires écrits avec la bibliothèque `Alcotest` que j'ai modifiés pour tester le comportement avec la LFU. Ces tests consistent en la création d'un cache, puis en l'ajout de valeurs, pour ensuite contrôler si les bonnes valeurs sont dans le cache. Enfin, j'ai écrit les spécifications formelles en Gospel dans la signature du module et je les ai testées avec Ortac.

Exemple de test unitaire avec Alcotest. On prend l'exemple de test de la fonction `replace` de l'algorithme LFU, lorsque le cache est plein. Dans un souci de convention et de lisibilité, on nomme le test comme ce que l'on cherche à tester, ici `discard`. On définit donc la fonction `discard`, qui prend un, en y déclarant une capacité de 10. Ensuite, on crée le cache avec l'appelle de Cache.v en donnant en paramètre la capacité.

```
let discard () =
    let cap = 10 in
    let t = Cache.v cap in
```

Ensuite, on ajoute des valeurs en faisant appel à une fonction externe `add_fresh_values`, qui va créer la liste des valeurs ajoutées dans le cache et plus particulièrement contenant leur clés. Cette fonction prend en paramètre le cache dans lequel ajouter des valeurs, ainsi que le nombre de valeurs. Ici nous ajoutons `cap` valeurs dans le but de remplir entièrement le cache.

```
let all_values = add_fresh_values t cap in
```

La fonctionnalité `match` de OCaml permet, à partir d'une liste, de se retrouver dans ce qu'elle contient. On y spécifie des cas devant être t on regarde ensuite à quoi ressemble la liste donnée. L'ensemble de tous les cas énoncés doit rassembler tous les cas possible. Ici, nous avons une liste de 10 éléments que nous comparons à deux motifs génériques : une liste remplie `h :: tl` et une liste vide `[]`. Le motif de liste remplie signifie que la tête de la liste (le premier élément) `h` est suivit `::` d'une autre liste `tl`.

Ici seul le cas de la liste pleine va être possible, car nous ajoutons plus haut 10 éléments dedans, donc si la liste est vide, le programme doit s'arrêter car c'est anormal. C'est l'appel à `assert false` qui va stopper l'exécution du programme en lançant une erreur.

```
match all_values with
| h :: tl -> (* ... *)
| [] -> assert false
```

Considérons une liste à 10 éléments et un cache `t` de capacité 10 plein. Nous allons dans un premier temps vérifier que les valeurs de la liste `tl` sont bien dans le cache. Pour cela, on fait appel à la bibliothèque `Alcotest` qui nous permet un affichage à l'exécution propre et pour chaque valeur de la liste, une recherche est faite.

```
match all_values with
| h :: tl ->
    List.iter (fun k ->
        Alcotest.(check key)
            "Added values are still present" k (Cache.find t k)) tl;
    (* ... *)
```

Ensuite, je crée une nouvelle clé k et je l'ajoute (en appelant `replace`) dans le cache qui est déjà plein.

```
match all_values with
| h :: tl ->
    (* ... *)
    let k = K.v () in
    Cache.replace t k k;
    (* ... *)
```

Ainsi, je vais pouvoir vérifier que chaque valeur est à sa place et que la taille du cache est restée de 10. Ce dernier point est vérifié avec l'appel à `Cache.size`, en précisant que la valeur que l'on doit recevoir est 10.

```
| h :: tl ->
    (* ... *)
    Alcotest.(check int) "Size is still ten" 10 (Cache.size t);
```

La valeur qui doit être supprimée est la moins fréquemment utilisée. Comme elles ont toutes les mêmes fréquences, c'est la première qui a été ajoutée qui va être supprimée, soit `h` de notre liste `all_values`. Nous testons une recherche de `h` dans le cache avec la fonction `Cache.mem` qui renvoie vrai ou faux si la valeur est dedans ou pas. Nous attendons un retour négatif, car la valeur a dû être supprimée. Enfin, nous vérifions de nouveau que les autres valeurs `tl` sont toujours présente dans le cache.

```
| h :: tl ->
    (* ... *)
    Alcotest.(check bool)
        "Unused value are removed" false (Cache.mem t h);
    Alcotest.(check bool)
        "Added value is still present" true (Cache.mem t k)
    (* ... *)
```

Tests avec Gospel et Ortac. J'ai ajouté des spécifications Gospel dans le fichier de signature du module de liste doublement chaînée dans le but de spécifier les contrats devant être toujours vérifiés et de marquer l'apparition de potentielles erreurs dans le module (avec les postconditions), ou aussi les utilisations dans la LFU ou dans la LRU (avec les préconditions). Les invariants sont des propriétés d'un type qui sont toujours vrais quelque soit la valeur de ce type.

J'ai formalisé des conditions que le type `'a t` doit toujours vérifier. Par exemple, la capacité doit toujours être positive et l'attribut `t.free` doit être compris entre -1 et la capacité.

```
(*@ with t
    invariant t.cap > 0
    invariant -1 <= t.free < t.cap *)
```

J'en ai aussi ajouté des plus complexes, concernant la valeur de `t.prev` et de `t.next`. Premièrement, de la même manière que `t.free`, je certifie que `t.prev` et `t.next` sont compris entre -1 et `t.cap`. Je spécifie aussi que quel que soit l'indice `i`, son suivant (respectivement précédant), si `i` n'est pas le dernier (*resp.* le premier), alors le précédent du suivant (*resp.* le suivant du précédant) est bien `i`.

```
(*@ with t
    invariant forall i. 0 <= i < t.cap -> -1 <= t.next.(i) < t.cap
    invariant forall i. 0 <= i < t.cap -> -1 <= t.prev.(i) < t.cap
    invariant forall i. 0 <= i < t.cap -> t.next.(i) <> -1
                -> t.prev.(t.next.(i)) = i
    invariant forall i. 0 <= i < t.cap -> t.prev.(i) <> -1
                -> t.next.(t.prev.(i)) = i *)
```

Ces contrats permettent ensuite de générer à l'aide d'Ortac le code qui va vérifier automatiquement ces invariants et contenir les arrêts nécessaires et un rapport d'erreur si une violation est détectée. Ainsi, lors de l'exécution du programme utilisant le module généré, le programme s'arrêtera directement si une condition est invalide, ce qui rend la recherche de bugs très facile puisque la fonction qui contient l'erreur est directement affichée.

4.6 Comparaison des stratégies

Pour comparer les algorithmes implémentés, j'ai écrit des benchmarks, c'est-à-dire un programme qui effectue des mesures du temps d'exécution des fonctions de caches.

4.6.1 Benchmarks avec `current-bench`

J'ai utilisé `current-bench`, une application développée à Tarides et fonctionnant sur GitHub pour permettre de construire des graphiques à partir de benchmarks, tout au long du développement de la bibliothèque (par exemple à chaque commit ou pull request).

Dans un premier temps, j'ai listé les différentes mesures que je voulais réaliser, par exemple le temps d'exécution des `add` et `find`. Pour cela, j'ai utilisé une trace des opérations réelles de cache réalisé dans Irmin par Tezos, afin de simuler une situation réelle.

Mesure du temps. Il existe plusieurs bibliothèques permettant d'obtenir des mesures de durées, par exemple la bibliothèque `Unix` et sa fonction `time`, ou la bibliothèque `Mtime`. J'ai choisi `Mtime` car `Unix.time` nous fournit l'heure du système, qui est moins précise, car elle contient aussi le temps d'exécution des autres processus tournant sur le système. Pour mesurer la durée d'exécution d'une fonction `f`, j'ai créé un compteur, valant l'heure courante. Je lance ensuite la fonction donnée puis je calcule la différence du premier compteur et du second, à la fin de l'exécution. Ce résultat est ensuite converti en millisecondes.

```
let time (f : unit -> unit) =
  let counter = Mtime_clock.counter () in
  f ();
  let span = Mtime_clock.count counter in
  Mtime.Span.to_ms span;
```

Pour obtenir des résultats plus précis, j'effectue les mesures plusieurs fois pour chaque fonction. La visualisation produite par `current-bench` permet ensuite d'afficher la moyenne et les valeurs extrêmes.

Exportation des résultats. Pour que le `current-bench` puisse rendre les temps d'exécution sous forme de graphiques visibles sur l'application, il faut que les données soient dans un format JSON. C'est un langage léger d'échange de données textuelles qui s'analyse facilement. Le fichier JSON attendu par `current-bench` possède un entête avec le nom du benchmark ainsi que des

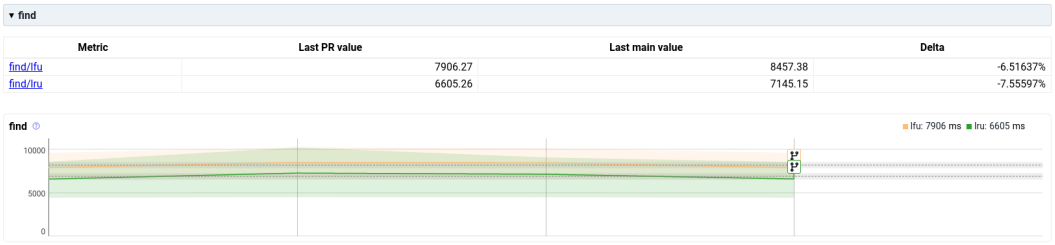

FIGURE 9 – Comparaison d'un find entre la LFU et la LRU

attributs optionnels, puis un champ `results` qui contient les mesures. Pour chaque test, on y met le nom du test et le résultat. L'application fait correspondre les entrées ayant comme même nom de tests afin d'en faire la moyenne automatiquement. Ci-dessous, un exemple de JSON qui indique le temps moyen pris par un add avec la stratégie LFU.

```
{"results": [{"name": "add", "metrics": [{"name": "add/Lru", "value": 5057.200029,
"units": "ms"},{"name": "add/LFU", "value": 5722.036765, "units": "ms"}]}]}
```

Intégration à GitHub. Pour lancer et envoyer les mesures à l'application il fallait réaliser un nouveau commit sur GitHub. L'application OCurent va s'exécuter lors de la réception des fichiers Ce dernier exécute le fichier nommé Makefile situé dans le projet envoyé. Celui que j'ai écrit télécharge la trace et exécute la fonction de mesure. J'aurais pu réaliser cela en local, mais il y a un serveur destiné au calcul de benchmarks, ce qui permet d'avoir des calculs plus précis, car il n'y a pas le temps d'exécution des autres processus de l'ordinateur tournant en parallèle, contrairement à faire cela en local.

Résultats. J'ai réalisé plusieurs tests de comparaisons avec la trace de irmin-pack.

- `add` add entre LFU et LRU : LFU plus rapide
- Comparaison de find entre LFU et LRU : LRU plus rapide
- Comparaison de find et add sur la LRU et LRU, avec différentes taille de cache (1 000, 10 000, 100 000)

Les résultats sont diversifiés et chaque algorithme de remplacement dans le cache à ses avantages et ses inconvénients. Il faut néanmoins noter que ces résultats sont partiels pour le moment, car ils ne permettent pas encore de tenir compte du temps d'accès au disque (seuls les accès au cache sont mesurés). Or ce sont ces mêmes accès que l'on cherche à éviter avec un cache et les taux de `hits` et `miss` jouent donc un rôle crucial dans l'utilisation des caches en conditions réelles.

4.6.2 Benchmarks avec `bechamel`

J'ai lancé d'autres types de benchmarks qui étaient déjà présents dans `cachecache`, ceux utilisés en particulier pour la LRU. Comme la LFU et la LRU on la même interface, il n'a pas été nécessaire de modifier le code pour obtenir des résultats. Ces benchmarks sont basés sur la bibliothèque `bechamel`, qui lance chaque test un grand nombre de fois afin de mesurer et éliminer les petites fluctuations dans les résultats. Elle est particulièrement adaptée à la mesure de temps d'exécution très courts comme c'est le cas ici et permet également de mesurer les allocations. J'ai mesuré les statistiques d'exécution pour les fonctions `replace` et `find`, pour différentes capacités de cache et pour chacune des implémentations présentes dans `cachecache` (LRU, LFU avec

première version des listes, LFU avec seconde version des listes). Les mêmes fonctions du module `Hashtbl` de la bibliothèque standard sont également ajoutées, pour comparaison.

Résultats. Les résultats obtenus par cette méthode sont présentés dans la figure 10. Ils suivent majoritairement le comportement attendu, en particulier :

— Les stratégies de caches sont à la fois plus lentes et allouent plus que les tables de hachage, puisqu'elles en utilisent une elles-même.

— La LFU est moins performante que la LRU, puisque chaque requête doit augmenter la fréquence d'utilisation de la clé, alors que la LFU ne doit mettre à jour ses données que lors d'un `replace`.

— La version finale des listes doublement chaînées rend la LFU plus efficace, à la fois en temps et en espace.

Fonction	Capacité	Module	Allocations (mots)	Temps (ns)
`find`	100	`Hashbtl`	5.4019	68.4427
		`Lru`	7.6276	108.7610
		`Lfu` (v1)	20.7870	223.2689
		`Lfu` (v2)	15.4268	205.7746
	1000	`Hashbtl`	5.4563	81.3451
		`Lru`	7.7109	117.1118
		`Lfu` (v1)	20.1884	267.0659
		`Lfu` (v2)	15.1620	197.0543
	10000	`Hashbtl`	5.7647	83.3585
		`Lru`	8.3761	172.9929
		`Lfu` (v1)	21.1961	276.4549
		`Lfu` (v2)	15.9194	235.6202
`replace`	100	`Hashbtl`	5.4369	118.5812
		`Lru`	12.4878	150.7466
		`Lfu` (v1)	28.9934	258.2246
		`Lfu` (v2)	20.9317	227.4149
	1000	`Hashbtl`	5.3610	163.0117
		`Lru`	12.5929	161.3663
		`Lfu` (v1)	29.1111	259.4852
		`Lfu` (v2)	21.0345	244.3692
	10000	`Hashbtl`	5.6854	170.8656
		`Lru`	3.4019	188.1846
		`Lfu` (v1)	30.2388	358.1491
		`Lfu` (v2)	21.8072	288.9505

FIGURE 10 – Les résultats ont été obtenus en lançant les benchmarks sur une machine avec un processeur i7-1165G7 @ 2.80GHz CPU, avec 16GB de RAM en utilisant le compilateur OCaml 4.14.0.

4.7 Hacking days

J'ai choisi d'implémenter l'automate cellulaire nommé la fourmi de Langton. C'est un jeu qui s'effectue par étapes, suivant des règles simples afin de passer d'une étape à la suivante. Cela se passe sur une grille, où chaque case représente une cellule, qui peut prendre deux états : mort ou vivant. La particularité de ce jeu, c'est que l'on place une (ou plusieurs) fourmi sur la grille, qui va se déplacer en modifiant les états des cellules la côtoyant. Cela suit ces simples règles :

— Si la fourmi est sur une case noire, elle tourne de 90° vers la gauche, change la couleur de la case en blanc et avance d'une case.

— Si la fourmi est sur une case blanche, elle tourne de 90° vers la droite, change la couleur de la case en noir et avance d'une case.

Ce petit projet s'est déroulé durant le début de mon stage, j'en ai profité pour y appliquer les techniques que je venais d'apprendre, notamment les modules, les foncteurs mais aussi l'environnement `dune` qui compile et lance les projets. J'y ai aussi ajouté des spécifications Gospel afin de me familiariser avec ce nouveau langage.

Troisième partie
Bilan

5 Apports du stage

Ce stage de fin de DUT a consisté en la création d'outils d'environnement pour la comparaison de d'algorithmes de mise en cache, ainsi qu'en l'implémentation de stratégies de remplacement de cache dans l'objectif d'en faire une bibliothèque nommée `cachecache` pour OCaml.

J'ai pu appliquer les connaissances que j'ai acquis durant ma formation à l'IUT d'Orsay, notamment dans la logique d'aborder un problème. En plus de développer mon autonomie aussi bien dans l'implémentation que dans la gestion du projet, j'ai amélioré mes compétences linguistiques par des réunions en anglais, mais aussi en rencontrant de nouvelles personnes anglophones.

C'est la première fois que je travaillais sur un projet open-source et le concept me plaît car tout le monde peut contribuer à celui-ci afin de le rendre meilleur.

J'ai aussi découvert un nouveau langage qu'est le OCaml, que j'apprécie particulièrement. Son utilisation s'accompagne d'outils de travail comme Dune ou Opam, qui sont très pratiques pour la gestion des bibliothèques. Ce stage a confirmé mon intérêt pour l'environnement fonctionnel et m'a motivé pour essayer d'autres langages similaire, ayant des objectifs différents, comme Rust. J'ai davantage compris l'utilité de Git, principalement pour la gestion de projets en équipe et je continue d'utiliser cet outil pour mes projets scolaires et personnels.

Enfin, j'ai découvert l'univers de l'entreprise ainsi que la gestion de projet dans une équipe de développeurs et compris les enjeux du monde professionnels.

5.1 Nouveaux langages

OCaml. J'ai approfondi les connaissances de base que j'avais dans ce langage, notamment en apprenant à approcher les problèmes de manière fonctionnelle. Enfin, j'ai découvert des paramètres comme `OCamLruNPARAMS=b` qui m'ont permis de trouver plus aisément des bugs en affichant la trace d'exécution précédent une exception.

Gospel. J'ai utilisé le langage Gospel, qui permet de réaliser des spécifications de programmes tout en étant simple d'utilisation. Il fonctionne grâce à des contrats que l'on écrit sous la signature de chaque fonction ou type d'un modules OCaml, qui décrivent le fonctionnement cette fonction ou les invariants du type. Je trouve cela pratique, car il permet à la fois à la compréhension de la fonction par quelqu'un de l'extérieur, mais aussi de déceler des bugs avec Ortac.

Markdown. J'ai réalisé quelques documents en Markdown, qui est un langage de balisage pour écrire du texte enrichi. Je l'ai utilisé à la fois pour rédiger de petits fichiers `README.md` qui contiennent en général une description du projet, ainsi que le manuel d'utilisation, mais aussi pour rédiger les rapports à réaliser chaque semaine pour récapituler et expliquer sur quoi j'ai travaillé la semaine passée. C'est un langage qui met en forme des lignes de texte simplement, que je pense utiliser dans le futur.

LaTeX. J'ai utilisé LaTeXpour écrire ce rapport car j'ai trouvé que c'était la bonne occasion d'essayer cet outil. C'est un langage de balisage, utilisé pour mettre en forme des documents.

J'utilise le site internet Overleaf afin de profiter d'une interface divisée en deux, d'un coté le code que j'écris et de l'autre mon document mis en forme en quasi temps réel.

5.2 Outils découverts

GitHub. C'est une plate-forme de stockage de code en ligne et de gestion de versions. Pour chaque utilisateurs, il est possible de créer des dépôts (nommés *repository*) qui représente des projets. Ils peuvent être privés ou publiques, mais comme nous travaillons en open-source, tout les dépôts sont publiques. Chacun d'entre eux a une branche principale appelée *main*, où le code est ajouté au fur et a mesure, en conservant tout les changements réalisés. Lors de projets, il est possible de créer plusieurs branches, afin de travailler à côté du `main` sans le modifier et une fois que les changements sont fonctionnels et prêts à être ajoutés à la branche principale, on effectue un *merge* sur la branche crée et la branche principale, un ajout de code à *main*. Sur ces branches, il est possible de créer une *Pull Request*, c'est à dire une demande d'ajout de code au dépôt. Cela crée une interface pour connaître les différentes modifications sur la branche. Il est possible d'ajouter des collaborateurs qui vont regarder le code mis en ligne afin d'émettre des idées, de traquer les problèmes ou de simplement laisser un commentaire, c'est ce qu'on appelle la `review`. Ensuite, il est facile de réaliser le `merge` une fois le travail terminé. J'avais déjà utilisé GitHub, notamment pour mes projets personnels. J'ai appris de nombreuses nouvelles fonctionnalités et je vais les utiliser dans le futur.

Opam. Abréviation de OCaml Package Manager, cet outil centralise toutes les bibliothèques pour OCaml et son utilisation. Ainsi, il est facile d'installer un nouveau package et ses dépendances.

Dune. Dans les projets, il existe un ou plusieurs fichiers `dune`. Ces derniers décrivent les besoins externes du projets, notamment les bibliothèques utilisées. Lors de la compilation du programme, nous appelons dune qui va lui, construire la commande pour exécuter le code OCaml, en y ajoutant toutes les bibliothèques utilisées. Lors de mes débuts, je n'utilisais pas cela, mais cela devient nécessaire lorsqu'un projet nécessite beaucoup de bibliothèques annexes.

Merlin. C'est un outil ajouté à l'éditeur de texte (Visual Studio Code, Emacs...), qui spécifie le type des variables lors de leurs utilisations. Je continuerai d'utiliser merlin, car cela m'a aidé de nombreuse fois lorsque je travaillais avec des types plutôt compliqués.

OCamlFormat. C'est un outil de mise en forme automatique de code. L'utiliser peut être perturbant dans un premier temps, car j'étais habituée a choisir moi même le style de mon code, mais il devient très utile dans les travaux d'équipes afin d'avoir un projet cohérent au niveau de la mise en forme.

Coq. Coq est un assistant de preuve, c'est a dire que ce logiciel permet d'écrire puis de vérifier des théorèmes, ou des assertions lors de l'exécution de programmes. Le langage utilisé pour décrire les éléments à vérifier est assez difficile à utiliser.

Ortac. Ce système génère à partir des contrats Gospel des assertions à vérifier lors de l'exécution d'un programme. Ensuite, un programme utilisant ce code généré sera stoppé en cas de non respect d'un contrat. Cela permet de facilement localiser des bugs, à condition que les contrats soient de qualité.

Why3. C'est un logiciel de preuve de programme, où le programme est écrit en WhyML, un langage très proche du OCaml. La plateforme va vérifier les conditions logiques entre chaque ligne du programme, pour déduire si les pré et post conditions sont toujours vraies. Ce logiciel, ainsi qu'Ortac m'ont aidée à vérifier la justesse de mes implémentations et sont bien plus simple d'utilisation que d'écrire les conditions en Coq.

6 Perspectives futures

Pour les quelques semaines qui restent à mon stage, je vais implémenter un dernier algorithme d'éviction nommé LFRU, qui résulte de la combinaison de la LFU et `Lru`. Je pourrais ainsi voir comment les tests évoluent et si une des stratégie se démarque dans les tests sur la trace d'Irmin.

Ce stage a confirmé mon intérêt pour le domaine de l'informatique et principalement dans le développement de logiciel et de bibliothèques afin de fournir davantage le langage. Je suis a présent persuadée de continuer mes études dedans, notamment dans un cursus d'ingénieur en informatique après mon DUT, afin de compléter mes connaissances actuelles et d'en acquérir de nouvelles.

L'ambiance générale dans l'entreprise Tarides est vraiment bonne, les locaux sont bien situés et très beaux, mais aussi les personnes avec qui j'ai pu travailler était très ouverts à m'aider si j'avais besoin, ou simplement à répondre à mes questions. En plus de cela, les différents projets démarrés par cette entreprise sont très intéressants, c'est pourquoi j'ai décidé d'y réaliser mon alternance durant 3 ans pendant l'école d'ingénieur.

Je suis très heureuse d'avoir réalisé mon stage à Tarides et d'avoir pu contribuer à un projet de grande envergure.

Glossaire

API Signifiant Iterface de Programmation d'Application, c'est un ensemble de définitions et d'implémentations de fonctions qui facilite l'utilisation de modules. 10

base de données Collection d'informations organisées afin d'être facilement consultables, modifiables et mises à jour. 7

bibliothèque Une bibliothèque, est un ensemble de fonctions regroupées afin d'être réutilisée par tout le monde sans avoir à les réécrire. 4, 6, 8, 10, 16

blockchain Mode de stockage et de transmission de données sous forme de blocs liés les uns aux autres et protégés contre toute modification.. 8

bug Défaut de conception d'un programme informatique à l'origine d'un dysfonctionnement. 7

compilateur Programme traduisant le code source écrit en langage machine, c'est-à-dire du binaire (suite de 0 et de 1). 4

cryptomonnaie Monnaie numérique en usage sur Internet, indépendante des réseaux bancaires et liée à un système de cryptage. 5

exception Une exception est un évènement qui apparaît pendant le déroulement d'un programme et qui empêche la poursuite normale de son exécution. Les exception sont lancée dans la structure `try` et peuvent être rattrapée par un `with` dans les langages OCaml.. 11, 15

foncteur En OCaml, un foncteur est une application prenant en paramètre un ou plusieurs module et en renvoyant. C'est l'équivalent des fonctions pour des valeurs. 7, 10, 11, 25

fonction Une fonction est une sorte de sous-programme qui effectue une tâche indépendante, pouvant être réutilisé à plusieurs reprises. 6

interface C'est l'ensemble des fonctions et types définis pour un module. 6

intégration continue L'intégration continue est une méthode de développement où chaque modifications entraîne un test vérifiant que toute les fonctionnalités sont encore fonctionnelle. 4

langage machine C'est une suite de 0 et de 1 nommé bits, qui est interprétée par le processeur d'un ordinateur. 7

liste doublement chaînée Liste dont chaque élément peut accéder à l'aide de pointeurs aux éléments positionnés immédiatement avant et après lui dans la liste. 8

module En OCaml, chaque morceaux de code forment des modules. Le nom de ces modules sont régis par le nom du fichier dans lequel il est écrit. 6–8, 10–12, 14, 26

multi-coeur Un coeur est un ensemble de circuit électronique intégrés au processeur, capable d'exécuter des programmes informatique de manière autonome. Lorsque l'on parle de multi-coeur, c'est que le processeur peut exécuter simultanément et en parallèle plusieurs programmes. Un programme fonctionnant en parallèle divise ainsi le temps d'exécution entre les différentes parties travaillant dessus. Un langage fonctionne pas forcément en multi-coeur, OCaml 5 a pour objectif de notamment fonctionner en parallèle. 4

mutable Un champs est mutable lorsqu'il peut être modifié, c'est-à-dire que la valeur qu'il contient peut changer. 6

n-uplet C'est un ensemble constitués de plusieurs valeurs non nommée, pouvant être de type différents. 14

open-source Code conçu pour être accessible au public. Ce qui inclu d'être visible, éditable et distribuable par n'importe qui. 7

pointeur Il contient l'adresse de la donnée en mémoire. 13–16

portable La capacité d'un langage ou d'un programme de pouvoir être adapté plus ou moins facilement en vue de fonctionner dans différents environnements d'exécution. 7

processus Iinstance d'un programme informatique en cours d'exécution. 11

RAM La mémoire RAM (Random Access Memory) est la mémoire vive de l'ordinateur. Elle accède directement aux données qu'elle stocke, ce qui la rends très rapide. 8

tables de hachage Structure de données qui permet une association clé–valeur, de complexité linéaire en recherche. 13

tests unitaire Test permettant d'évaluer des petites parties d'un programme ou d'un projet. 8

type Définie la nature de la valeur, cela peut être des types classiques (entier, booléens, chaîne de caractères...) mais aussi des types crée par l'utilisateur. 7, 8, 12–14, 17, 19, 27

vecteur Dans certains langages incluant OCaml, un vecteur est l'appellation d'un tableau redimensionnable. 18

Table des figures