Safe concurrent abstractions for multicore OCaml

Rapport de stage L3, sous la direction de Gabriel Radanne et Ludovic Henrio, équipe CASH au Laboratoire d'informatique du parallélisme (Lyon).

MARTIN ANDRIEUX, École Normale Supérieure de Rennes, France

Le but de ce stage est de réaliser une bibliothèque d'acteurs pour le langage OCaml. Nous présentons dans ce rapport les principes de l'abstraction des acteurs pour la programmation parallèle, et détaillerons ensuite les éléments nécessaires à l'implémentation d'une telle bibliothèque. Nous évaluerons finalement les performances de nos acteurs à travers divers exemples. Nous expliquerons également les quelques nouveautés du langage OCaml 5 que nous utiliserons pour notre bibliothèque.

1 INTRODUCTION

1.1 Motivations

La cinquième version majeure du langage OCaml est la première à permettre la programmation multicoeurs. On y trouve une interface classique au parallélisme à base de *fork* et *join*. Ce système est efficace, mais offre peu de garanties aux programmeurs. Raisonner sur un programme parallèle est délicat, beaucoup de propriétés des programmes séquentiels sont perdues, et les supposer vraies peut lever des erreurs à runtime.

Le problème le plus commun est celui de la mémoire mutable partagée. Si deux threads différents ont accès à la même donnée, ils sont susceptible de la modifier en même temps. Au mieux, les deux threads récupèrent deux versions différente de la même donnée. Au pire, l'ordre d'écriture brise un invariant du programme, ce qui peut causer divers bugs difficiles à identifier. La méthode classique pour éviter ce problème est d'utiliser des verrous (mutex) afin d'éviter deux accès simultanés à la même donnée ; mais cela entraîne des chutes de performances et est, une fois de plus, difficile à déboguer en cas de deadlock. Il est de plus difficile de programmer avec ce système : les programmeurs doivent savoir exactement quel thread peut écrire sur quelle donnée et quelles données sont partagées.

Nous souhaiterions donc une autre interface au parallélisme, gérant différemment le partage de la mémoire.

1.2 Approche et contribution

Nous présentons dans ce document le modèle des acteurs [Agha 1986], où différents opérateurs (appelés *acteurs*) communiquent par passage de messages et possèdent leurs propres mémoires. Ces acteurs fonctionnent de manière asynchrone en *promettant* des résultats qu'ils calculent quand bon leur semble. Lorsque l'on programme avec des acteurs, on ne partage plus la mémoire mais l'acteur la possédant. Pour modifier la mémoire, il faut envoyer un message à l'acteur, qui effectuera la modification. Un programme acteur ne nécessite donc ni fork ni join ni mutex.

Notre but est d'écrire une bibliothèque d'acteurs pour OCaml facile d'usage et performante, le tout en tirant partie des nouveautés apportées par la dernière version.

1.3 État de l'art

Les acteurs ne sont pas une nouvelle manière de faire du parallélisme, une présentation du modèle est faite dans [Agha 1986]. Un langage d'acteurs minimal y est également décrit. Un

Author's address: Martin Andrieux, Département d'Informatique, École Normale Supérieure de Rennes, Bruz, 35170, France, martin.andrieux@ens-rennes.fr.

tel langage permet de rendre compte des possibilités des acteurs, mais ne sera jamais utilisé à grande échelle. Des bibliothèques telles que *akka* (https://akka.io) pour Java rendent accessible la programmation orientée acteurs. Rien de semblable n'existe pour le moment en OCaml.

Les acteurs sont aussi implémentés dans des langages compilés, comme Encore, où on retrouve un système de promesses et des appels optimisés tels que `forward` [Fernandez-Reyes et al. 2018]. Les garanties d'Encore reposent sur un système de types riche, capable de traquer l'utilisation des attributs locaux d'un acteur. Il n'est pas possible d'utiliser un tel système de type en OCaml. Nous devrons donc utiliser d'autres moyens pour garantir l'isolation mémoire.

2 CONTEXTE

Nous détaillons dans cette section les concepts de *promesses* et d'*acteurs*, afin de familiariser le lecteur avec le sujet. Nous expliquons également les quelques nouveautés d'OCaml 5 que nous utilisons dans notre bibliothèque.

2.1 Promesses

2.1.1 Analogie. Considérons deux individus Alice et Bob : au fil de leur discussion, Alice promet à Bob de revenir avec un *laissez-passer A38*. Elle donne alors un morceau de papier à son interlocuteur sur lequel il est écrit quelque chose du genre « Moi, Alice, promet de revenir avec un *laissez-passer A38* », et s'en va enquérir ce fameux *laissez-passer*. Bob se retrouve seul avec le papier d'Alice ; il pourrait rester là, passivement, en attendant que sa camarade accomplisse sa promesse, mais il décide d'aller faire autre chose et de revenir plus tard (Bob est très occupé). Lorsqu'il revient, deux cas de figures sont possibles :

1 – Un *laissez-passer A38* est là, déposé par Alice, qui est aussitôt repartie. Heureux de lui avoir fait confiance, Bob récupère le *laissez-passer* et retourne à ses occupations.

2 – Pas de *A38* ni d'Alice à l'horizon, Bob attend patiemment le retour de son amie. Lorsque cette dernière arrive, elle tient un *A38* dans ses bras. Elle le donne à Bob et se séparent.

2.1.2 Explications. Dans notre exemple :

— Nos individus sont deux threads, capables de communiquer et de réaliser des actions en parallèle.

— Le morceau de papier est une *promesse*, cette promesse est une valeur de première classe, et peut donc être manipulée comme n'importe quelle autre donnée du langage [1]. Elle représente le futur *laissez-passer*, délivré par Alice.

— Les « occupations » de nos individus représentent des calculs : les threads peuvent effectuer d'autres calculs avant d'attendre une promesse.

— Enfin le *laissez-passer A38* est le résultat du calcul. Il peut être disponible après ou avant l'accès à la promesse, impliquant (ou non) une attente de Bob.

En bref, une promesse peut être vue comme une boîte, initialement vide, pouvant être partagée librement entre différents threads, qui sera remplie avec le résultat d'un calcul par l'un et lue par les autres.

1. Elle peut donc être partagé, passée en paramètre à des fonctions etc.

2.1.3 Contextes d'utilisation. Les promesses forment une interface agréable aux fonctions asynchrones [2]. Elle sont par exemple un élément central du langage Multilisp [Halstead 1985]. Les promesses étaient déjà massivement utilisées en OCaml pre-multicore, notamment dans les bibliothèques *Lwt* et *Eio* [3]. Nous donnons figure 1 le code correspondant à notre exemple introductif. On y utilise les fonctions suivantes :

— `Promise.async : (unit -> 'a) -> 'a Promise.t` exécute la fonction passée en argument de manière asynchrone et renvoie directement une promesse du résultat.

— `Promise.get : 'a Promise.t -> 'a` renvoie la valeur de la promesse une fois cette dernière accomplie.

```
let promised_A38 = Promise.async (fun () ->
  alice_does_stuff (); "A38") in
bob_does_stuff ();
Promise.get promised_A38
```

FIG. 1 – Traduction OCaml de l'exemple introductif

Certains langages utilisent la notion de futur qui est très proche d'une promesse. Un futur est automatiquement rempli à la fin d'un calcul alors qu'une instruction spécifique doit être appelée pour remplir une promesse (il faut alors s'assurer que la promesse est remplie une et une seule fois). Dans les faits, nous utilisons des promesses pour exposer des futurs, nous continuerons donc d'utiliser le terme promesse dans ce rapport.

2.2 Acteurs

2.2.1 Définition et mécanismes. Un acteur est, à l'instar de la programmation orientée objets, un ensemble de methodes et d'attributs. Un acteur est exécuté dans son propre thread, et communique avec le reste du monde par passage de messages. Chaque acteur est doté d'une *mailbox* : une file contenant les messages reçus, en attente de traitement. Envoyer un message à l'acteur revient donc à pousser un message dans la mailbox, ce qui se fait de manière asynchrone.

Un acteur pourrait demander à un autre acteur d'effectuer un calcul, et de lui renvoyer la réponse par message. Mais cela rend la programmation difficiles, car l'acteur qui fait le calcul doit connaitre l'identité de celui qui le demande, il doit lui envoyer un message de réponse et l'acteur qui a lancé le calcul doit savoir réagir au message qui contient la réponse. Une manière plus « naturelle » de programmer (en tout cas plus commune) est de stocker quelque chose faisant office de résultat pour pouvoir le manipuler. Il est donc logique que nos acteurs renvoient des promesses. Du point de vue des programmeurs, les acteurs se comportent comme tel :

— l'appelant fait un appel de méthode à un acteur `A` et reçoit immédiatement une promesse.

— l'acteur `A` reçoit un message, il crée directement une nouvelle promesse `p` dont il envoie une référence à l'appelant. Lorsque le message arrive au bout de la mailbox, il est exécuté et la promesse est remplie avec le résultat.

2. Un appel de fonction asynchrone n'interrompt pas le fil d'exécution courant.

3. Pour plus d'informations sur Lwt et Eio, se référer à https://ocsigen.org/lwt/latest/api/Lwt et https://github.com/ocaml-multicore/eio

Ce procédé est illustré figure 2. Cette représentation est idéale, car elle suppose que les méthodes peuvent être exécutées « en une fois », ce qui n'est pas toujours possible [4].

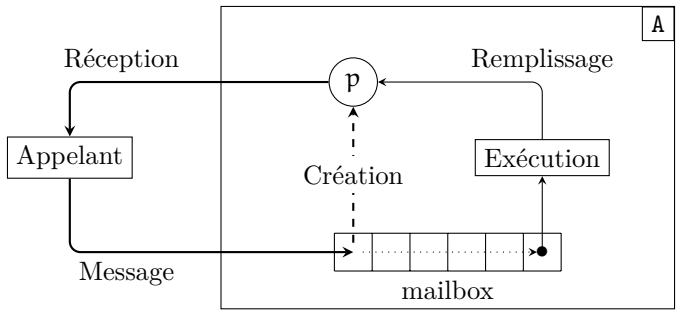

FIG. 2 – Fonctionnement simplifié d'un acteur, les flèches épaisses représentent les actions effectuées lors de l'envoi du message

Nous donnons ci-après un exemple d'acteur comme ils sont écrits dans notre bibliothèque :

```
let a = object%actor
  val mutable x = 0
  method set n = x <- n
  method get = x
  method multiply n = x * n
end
```

Ici, `object%actor` définit un nouvel acteur, une extension de syntaxe ajoute du code à la compilation, permettant aux programmeurs de se concentrer sur le code de l'acteur plutôt que sur l'implémentation de ce dernier. À l'intérieur du `object%actor ... end`, les programmeurs peuvent utiliser deux types de définitions :

— `val` définit un attribut local à l'acteur. Cet attribut est inaccessible en dehors de l'acteur, mais peut être utilisé dans les différentes méthodes. Il est possible d'utiliser des attributs mutables en ajoutant le mot clé `mutable`. On peut alors modifier le champ avec la syntaxe `x <- value`, comme dans la méthode `set`.

— `method` défini une méthode publique de l'acteur. On appelle des méthodes avec l'opérateur [5] `(#!)`. Par exemple, `a#!get : int Promise.t` envoie le message `get` à l'acteur `a` (et retourne une promesse d'entier). Le corps d'une méthode est évalué à chaque appel, qu'elle ait des arguments ou non, ce qui diffère de la syntaxe OCaml habituelle (où une fonction sans arguments est une valeur dont le corps ne sera exécuté qu'une unique fois).

2.2.2 Coopération. Les lecteurs sont en droit de se demander ce qu'il se passe si un acteur se fait une promesse à lui-même. S'il attend la promesse, celle-ci ne pourra jamais être accomplie [6], car l'acteur sera occupé à attendre et pas à accomplir la promesse. Nous donnons ci-dessous un exemple d'une telle situation : l'appel à `a#!bar` ne termine pas, car `a` s'envoie un message `foo` et attend, il ne traitera donc jamais `foo`.

4. Nous revenons sur ce point en section 2.2.2.
5. Il ne s'agit en réalité pas d'un opérateur, cette construction n'a pas de type.
6. Plus généralement, cela advient lorsque le graphe de dépendance des promesses admet un cycle une fois chaque promesse identifiée avec son acteur

```
let a = object%actor (self)
  method foo = 42
  method bar = Promise.get (self#!foo)
end
```

Il nous faut donc un moyen de *coopérer* au sein d'un acteur, c'est à dire de laisser d'autres messages s'exécuter car nous avons besoin de leurs résultats. Dans notre exemple, cela consiste à arrêter l'exécution de `bar` au profit de `foo`. Nous ajoutons donc, en plus de `get`, la fonction `await`, qui coopère si la promesse passée en paramètre n'est pas accomplie. Cette coopération se traduit par un retour du calcul courant dans la fille des messages, les autres processus peuvent alors être exécutés par l'acteur et résoudre les promesses en attente.

Nous aurions donc pu (et dû) écrire :

```
let a = object%actor (self)
  method foo = 42
  method bar = Promise.await (self#!foo)
end
```

2.2.3 Autres appels. Notez que dans cet exemple, `foo` crée une promesse qui est tout de suite attendue pour être ré-emballée dans la promesse de `bar`. Or, l'allocation de promesses n'est pas gratuite, et il faut chercher autant que possible à réduire leur nombre.

Nous disposons pour cela de la construction *forward* [Fernandez-Reyes et al. 2018]. Forward permet de *déléguer* l'accomplissement de la promesse à un autre acteur (ou à soi-même). Ainsi, dans l'exemple précédent, `bar` pourrait demander à `foo` de remplir sa promesse directement. Les schémas mémoires associés à ces deux appels sont représentés figure 3. On y voit en gris la promesse évitée par l'appel de `forward`. Notez que `forward` stoppe le calcul de l'appelant afin d'éviter un double remplissage de promesse. L'appel à `forward` doit donc être *unique* et *terminal*.

FIG. 3 – À gauche, l'appel asynchrone avec deux promesses. À droite, l'appel `forward` remplissant directement la promesse de `bar`.

De plus, comme l'appel est fait sur `self`, nous aurions pu utiliser l'*appel synchrone*, correspondant à un appel de méthode « classique », qui ne crée ni de message ni de promesse, et est donc plus rapide dans cette situation. L'appel synchrone est bien plus flexible que `forward`, il peut être utilisé à de multiples reprises dans une même méthode. L'appel synchrone n'évite la création de promesse uniquement lorsqu'il est appelé sur `self` ; dans les autres cas, il sera remplacé par un `p = send(...); get p`. La syntaxe de ces deux appels est donnée figure 4.

2.3 OCaml : Domaines et effets

Nous expliquons dans cette sous-sections les quelques nouveautés de la cinquième version d'OCaml que nous utilisons dans notre bibliothèque.

```
(* Forward *)
let a = objec%actor (self)
  method foo = 42
  method bar = self#!!foo
end
```

```
(* Sync call *)
let a = objec%actor (self)
  method foo = 42
  method bar = self#.foo
end
```

FIG. 4 – Acteur a avec `forward` et l'appel synchrone

2.3.1 Domaines. En OCaml, un *domaine* est un thread système disposant d'une mémoire locale et (a fortiori) de son propre *ramasse-miettes* (garbage collector). Chaque domaine créé (avec `Domain.spawn`) doit être terminé explicitement (avec `Domain.join`), sous peine de comportement indéterminé.

Ainsi, les acteurs présentés précédemment sont idéalement [7] exécutés seuls dans leur domaine. Un calcul effectué dans un domaine dispose également d'une mémoire locale et inaccessible des autres domaines, nommée *Domain Local Storage* (DLS).

2.3.2 Système d'effets. OCaml 5 propose, en plus de la programmation parallèle, un système d'effets. Un effet peu être vu comme une exception : on lève un effet avec `perform` (équivalent de `raise`), et on les traite dans un *handler* (équivalent d'un `try with`).

Seulement, en plus de l'effet, le handler récupère la continuation du calcul ayant lancé l'effet. Il peut alors relancer (ou *continuer*) la continuation avec la valeur de son choix, ou bien lui renvoyer une exception (*discontinuer*). Récupérer la continuation permet de retarder son relancement, on peut donc la stocker pour la reprendre plus tard, à condition de continuer (ou discontinuer) chaque continuation exactement une fois.

Une utilisation non triviale des effets est présentée [8] figure 5 page ci-contre : les deux fonctions `sender` et `receiver` tentent de communiquer de l'information. Elle passent par la fonction `step`, qui les exécute sous un handler d'effets. L'idée est d'interrompre les deux calculs sur leur points de communication respectifs afin de pouvoir les relancer avec l'information récupérée. On a `step sender = (Send 42, fun () -> ())` et `step receiver = (Wait, fun n -> print_int n)`. La fonction `main` peut alors continuer `receiver` avec `42`, l'entier envoyé par `sender`.

3 UNE BIBLIOTHÈQUE D'ACTEURS EN OCAML

Dans la section précédente, nous avons présenté des exemples de code acteurs en OCaml. Ces exemples utilisent la bibliothèque implémentée pendant ce stage. Nous nous intéressons désormais à certains détails techniques de l'implémentation de ladite bibliothèque.

3.1 La bibliothèque en détails

Nous détaillons dans cette partie l'ensemble des fonctionnalités implémentées dans la bibliothèque et sur lesquelles nous nous arrêterons plus longuement dans la suite de ce rapport.

Définir un acteur se fait comme expliqué dans la section 2.2. Une extension de syntaxe est ensuite appliquée à l'ensemble du code, à la compilation. Cette extension ajoute notamment le système d'appels asynchrones, permettant aux méthodes de renvoyer des promesses.

7. Nous verrons que cela n'est en réalité pas possible, dû à des contraintes du langage.
8. Pour plus de lisibilité, cet exemple utilise une syntaxe ficitive pour les effets. OCaml ne fournit pour le moment qu'une interface utilisant des fonctions.

```
effect Send : int -> unit
effect Recv : int

type Status =
 | Sended of int
 | Wait

let sender () =
  perform (Send 42)

let receiver () =
  let n = perform Recv in
  print_int n

let step f =
  try f () with
  | Send n, k -> Sended n, k
  | Recv, k -> Wait, k

let main () =
  match (step sender), (step receiver)
    with
  | (Sended n, k), (Wait, k') ->
    continue k ();
    continue k' n
  | _ -> failwith "Synchro error"
```

FIG. 5 – Un exemple d'utilisation des effets en OCaml

Nous ajoutons pour cela des méthodes supplémentaires aux acteurs faisant appel à celles initialement définies dans l'objet. Ces méthodes ne sont appelables que par le biais des opérateurs `(#!)`, `(#.)` et `(#!!)` décrits section 2. Un exemple de code généré par l'extension de syntaxe sera donné et expliqué section 3.6 après l'explication des détails techniques nécessaires à sa compréhension.

L'objet résultant est un simple objet classique d'OCaml, avec des méthodes et des attributs supplémentaires, emballé dans un type `Actor.t`, pour le différencier clairement des objets classiques.

Les deux fonctions `Promise.get` et `Promise.await` permettent de récupérer le résultat d'une promesse. La première est bloquante tandis que la seconde coopère, comme expliqué en section 2.1. `Promise.is_ready p` renvoie `true` si, et seulement si, `p` est accomplie.

En plus de l'appel asynchrone classique, nous avons implémenté `forward` et l'appel synchrone. Forward prend deux forme : la première permet de faire un appel de méthode en déléguant le remplissage de la promesse courante, il s'agit de `(#!!)` ; la seconde est `Actor.forward : 'a Promise.t -> 'a`, qui « unifie » la promesse en argument avec la promesse courante et interrompt le calcul. Cette fonction ne retourne pas (elle lève une exception), le type de retour `'a` n'est précisé uniquement pour des contraintes de typage. Deux promesses unifiées seront remplies au même moment avec la même valeur, comme s'il s'agissait d'une unique promesse.

Nous garantissons de plus que les attributs mutables d'un acteurs sont stockés dans la mémoire de son domaine (DLS) et qu'un acteur externe ne peut observer la mutabilité qu'au travers des messages.

Un programme utilisant notre bibliothèque consiste donc en la déclaration et définition de toute une série d'acteurs, puis d'un acteur `Main` : application de la fonction principale (`main`) à `Actor.Main.run`. De nombreux exemples peuvent être trouvés sur le dépôt suivant : https://github.com/Marsupilami1/actors-ocaml.

3.2 Promesses

3.2.1 Implémentation. Les promesses sont implémentées de la manière suivante dans notre bibliothèque [9] :

```
type 'a status =
  | Fulfilled of 'a
  | Failed of exn
  | Empty of ('a -> unit) list
type 'a t = 'a status Atomic.t
```

Une promesse est une référence atomique sur un statut, cette atomicité est importante pour garantir la thread-safety. Une promesse est donc :

— Soit vide, et accumule des *callbacks* qui seront exécutés lors du remplissage de la promesse. Ces callbacks permettent entre autre d'écrire des fonctions comme `map` et `bind`, fournissant une interface monadique aux promesses.

— Soit accomplie, et peut contenir le résultat du calcul ou l'exception qu'il a éventuellement lancé.

Les fonctions de lecture de promesses sont simples, car la logique est placée dans les handlers d'effets. La fonction `get` est par exemple définie comme montré figure 6 (la fonction `await` est semblable, mais utilise l'effet `Await`) :

```
effect Get :
  'a Promise.t -> 'a
let get p =
  match Atomic.get p with
  | Filled v -> v
  | Failed e -> raise e
  | Empty _ ->
      perform (Get p)
```

```
...
| Promise.Get p -> Some (
    fun (k : (a, _) E.continuation) ->
      while not (Promise.is_ready p) do
        Domain.cpu_relax ()
      done;
      E.continue k (Promise.get p)
  )
...
```

FIG. 6 – Fonction get (à gauche) accompagnée du traitement de l'effet éponyme (à droite)

Les effets `Await` et `Get` sont destinés à être récupérés par les ordonnanceurs des acteurs, qui décident du mécanisme d'attente (comme bloquer sur `Get`). On trouve par exemple dans l'ordonnanceur de l'acteur principal le second code de la figure 6

3.2.2 Ordonnanceur simple. Nous présentons ici un premier ordonnanceur simple tirant partie du système d'effet précédemment introduit. Le but de cet ordonnanceur est de permettre la coopération entre processus comme expliqué section 2.2. Chaque acteur possède un tel ordonnanceur, qui ne traite que ses processus.

Plutôt que de considérer une mailbox dans laquelle arrivent des messages, nous optons pour une file de *processus*. Un processus est constitué d'une promesse à remplir, ainsi que du

9. Malgrès l'existence de promesses dans d'autres bibliothèques d'OCaml, nous avons fait le choix de les réimplémenter entièrement. Outre l'intérêt pédagogique du stage, nous disposons d'un contrôle beaucoup plus fin sur nos promesses. Il est tout de même intéressant de chercher une certaine compatibilité avec les autres bibliothèques (plus complètes, notamment pour la gestion des entrées/sorties) en vue d'une unification.

calcul associé. Envoyer un message à l'acteur consiste donc à pousser un nouveau processus dans la file, constitué d'une promesse fraîche et de l'appel de méthode correspondant.

Nous exécutons chaque calcul dans un handler d'effets, à la manière de l'exemple de la figure 5 page 7. Ce handler récupère entre autre l'effet `Await`, il doit donc être capable de suspendre un calcul.

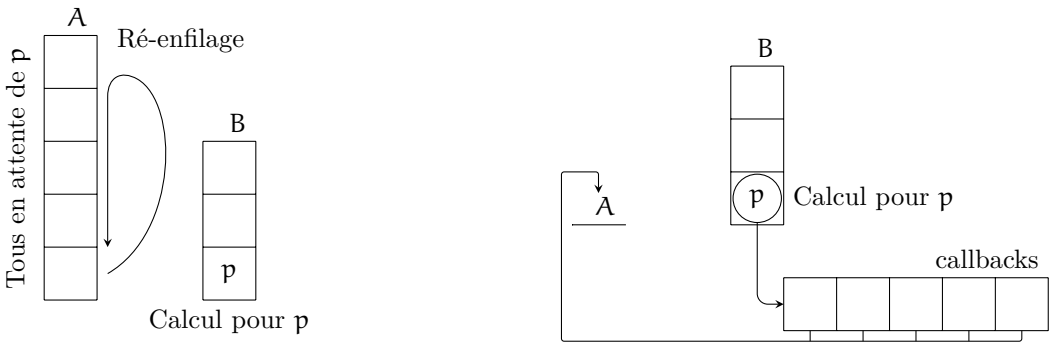

FIG. 7 – Attentes active (à gauche) et passive (à droite) de la promesse p

Nous pouvons considérer deux stratégies de suspension des calculs, toutes deux illustrées figure 7 :

1 – La première, l'attente *active*, consiste à repousser directement le processus dans la file. Cela permet aux autres processus de l'acteur d'être exécutés et d'éventuellement accomplir la promesse qui a suspendu notre premier calcul. Cette solution a le mérite d'être facile à implémenter, mais présente un inconvénient majeur : les processus n'ont pas la garantie que la promesse qu'ils attendent sera résolue lors de leurs prochaines exécutions. Cela peut entraîner une consommation importante de ressources, car nous devrions tester l'accomplissement de la promesse à chaque ré-exécution du processus.

2 – La seconde stratégie, l'attente *passive*, consiste à retarder l'ajout du processus dans la file à l'aide d'un callback sur la promesse ayant lancé l'effet : ainsi, si le processus est ré-exécuté, c'est qu'il a été poussé dans la file de l'ordonnanceur et donc que la promesse qu'il attend à été résolue. Cela règle le problème de la consommation de ressources, car l'acteur peut bloquer si sa file est vide.

Un calcul pourrait vouloir coopérer de lui-même, sans avoir à lire le contenu d'une promesse. Nous ajoutons pour cela l'effet `Yield`, qui remet le processus en bout de file (comme le faisait l'attente active). Cet effet permet d'écrire avec aisance des structures de contrôle complexes comme `wait_for`, qui attend qu'une condition soit vérifiée et dont l'implémentation est donnée figure 8.

3.3 Réduction du nombre de promesses

Nous avons identifié section 2.2.3 deux cas où la création d'une promesse est évitable et souhaitons ajouter les deux appels présentés à notre bibliothèque. Nous pourrons ensuite comparer ces différentes optimisations à travers divers exemples. Les résultats de ces benchs sont présentés section 4.

```
let wait_for (condition : unit -> bool) =
  while not (condition ()) do
    perform Yield
  done
```

FIG. 8 – Code de la fonction `wait_for`

3.3.1 Forward. Nous intégrons la fonction `forward` à notre ordonnanceur sans difficultés grâce au handler d'effets : lorsqu'un calcul fait un appel à `forward`, on lance l'effet `Forward` qui suspend le calcul et redonne la main à l'ordonnanceur. Ce dernier peut alors *discontinuer* l'appelant et lancer l'appelé avec la promesse courante. Cette opération est illustrée figure 9. On y voit l'appelant être remplacé par l'appelé, devant remplir la même promesse.

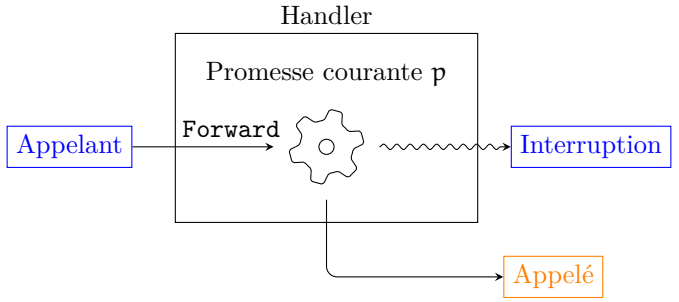

FIG. 9 – Handler d'effets traitant `Forward`

Notez que l'utilisation de forward nécessite l'existence d'une promesse courante et donc d'un appel de méthode. Utiliser `forward` dans le fil d'exécution principal lève une erreur à runtime (car l'effet `Forward` n'est pas récupéré).

3.3.2 Appels synchrones. Les appels synchrones ne passent pas par l'ordonnanceur. Ils sont simplement traduits par un test d'appartenance au même domaine, suivit de l'appel adapté (soit appel direct, soit création de promesse). Cela implique que les acteurs connaissent le domaine dans lequel ils sont exécutés. Cette information prend la forme d'une méthode synchrone `domain` ajoutée aux acteurs par l'extension de syntaxe.

3.4 Réduction du nombre de domaines

Créer un domaine est une opération coûteuse, et leur nombre est limité à 128 dans la version courante d'OCaml 5. Nous souhaitons donc avoir la possibilité d'exécuter plusieurs acteurs dans un même domaine, deux choix s'offrent à nous :

1 – Les acteurs sont répartis uniformément dans un pool de threads, suivant leur ordre de création. De l'extérieur, tout ce passe comme si les acteurs étaient seuls dans leur domaine.

2 – Le choix du regroupement des acteurs est laissé aux programmeurs, qui peuvent regrouper les acteurs intelligemment.

L'avantage de la seconde méthode est qu'elle encourage la coopération entre les acteurs d'un même domaine. Cela permet d'utiliser d'avantage d'appels synchrones et simplifie

l'ordonnancement. Cependant, elle ne limite en rien le nombre de domaines utilisés, et demande aux programmeurs de bien connaître la bibliothèque qu'ils utilisent. Nous avons donc choisi d'implémenter la première méthode, qui laisse l'interface de la bibliothèque inchangée. Cela permet entre autre de lancer la totalité des threads au démarrage du programme, ce qui réduit considérablement le temps de création d'un acteur.

Nous pourrions simplement pousser les messages de tous les acteurs dans une même file et laisser l'ordonnanceur inchangé, nous ferions alors l'hypothèse que deux acteurs dans un même domaine se comportent comme un unique acteur possédant les méthodes et les attributs des deux premiers. Mais cette représentation des acteurs est problématique :

— Nous ne voulons pas privilégier un acteur parmis l'ensemble des acteurs d'un domaine. En particulier, un acteur recevant beaucoup de messages ne doit pas empêcher ses voisins d'exécuter leurs processus. On cherche une forme d'équité entre les acteurs d'un même domaine.

— Les opérations bloquantes d'un acteur ne doivent pas bloquer les autres acteurs du domaine, car les programmeurs ne savent pas quels acteurs partagerons un domaine, et doivent pouvoir considérer qu'il sont totalement indépendants.

Pour ces raisons, nous instaurons un second niveau d'ordonnancement pour les acteurs. Il y a désormais dans chaque domaine une file d'acteurs contenant chacun une file de processus. Les acteurs sont traités dans l'ordre et exécutent un message à la fois comme illustré figure 10, où sept acteurs ont été répartis sur deux domaines. En supposant qu'aucun message n'est ajouté au cours de l'exécution du programme, l'ordre de traitement des messages sera celui indiqué.

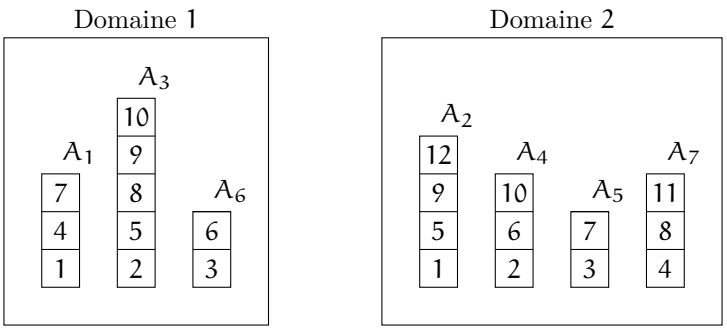

FIG. 10 – Partage d'un domaine par des acteurs

3.5 Gestion de la mémoire (attributs locaux)

L'état local des acteurs ne doit pas être accessible depuis l'extérieur, car cela casserait l'abstraction et les propriétés associées. Nous souhaitons donc assurer un maximum d'isolation des attributs locaux et vérifier qu'ils ne sont pas partagés.

Tout d'abord, remarquons que les attributs locaux non mutables ne représentent aucun danger et peuvent être partagés librement, comme n'importe qu'elle autre constante. Ils ne subiront donc aucun traitement particulier de notre part. Les attributs mutables, eux, nécessitent une plus grande attention.

Afin d'assurer que les attributs mutables ne puissent être modifié que par l'acteur auquel ils appartiennent, nous les encapsulons dans des DLS. Les données stockées dans un DLS

(Domain Local Storage) ne sont accessible que dans le domaine où elle ont été définies, il est donc impossible de modifier un attribut en dehors du domaine de l'acteur. Cette encapsulation est automatique, l'extension de syntaxe récupère tous les champs `val mutable x = ...` et modifie toutes les occurrences de `x` dans le corps des méthodes. Il faut pour cela résoudre les noms, car d'autres variables nommées `x` pourraient faire référence à un `let` local. Notez que la résolution de nom n'est pas habituellement disponible dans une extension de syntaxe, et est notoirement complexe en OCaml. Nous avons développé une bibliothèque utilitaire qui permet de déléguer cette tache aux typeur OCaml et récupère un AST annoté avec les noms souhaités résolus.

Des programmeurs non-avertis ou malintentionnés pourraient écrire le code ci-dessous :

```
let a = object%actor
  val mutable x = 0
  method setter = let f n = x <- n in f
end
```

La méthode `setter` permet à un acteur externe de modifier `x`, à condition qu'il soit exécuté dans le même domaine (car `x` est dans un DLS). Une fois de plus, cela casse l'abstraction, car l'acteur n'est plus le seul à pouvoir modifier ses attributs. Le même problème se pose pour une lecture de `x` dans la clôture : un changement de la valeur de `x` dans l'acteur ne doit pas changer le comportement de la clôture. Pour ces raisons, nous interdisons les clôtures capturant l'état local d'un acteur, et le code ci-dessus lève l'erreur suivante à la compilation :

```
Closures cannot capture internal mutable state, you may want to
    use something like:
|
| let a = x in fun _ -> ... a ...
|
Instead of:
|
| fun _ -> ... x ...
|
```

Cela est encore une fois vérifié à la compilation par notre extension de syntaxe, et utilise la résolution des noms précédemment calculée. Il est important de remarquer que nous n'avons accès qu'à des critères syntaxiques avec éventuellement la connaissance des occurrences d'une variable initialisée dans un `let`. Cela ne permet pas des jugements fins du type « Cette clôture capture de l'état local, mais n'est exécuté que dans le cœur de l'acteur, il n'y aura donc pas de problèmes de fuite d'état ». De tels raisonnements sont possibles dans le langage *Encore*[10], qui dispose d'un système de type plus riche que celui d'OCaml, ou en Rust via le mécanisme d'ownership.

Les appels à `forward` dans les clôtures sont également sources de problèmes :

```
let a = object%actor (self)
  method bar = 42
  method foo = let f () = self#!!bar in f
end
```

Dans cet exemple, quelle promesse est censée être remplie par l'appel de `forward` ? Celle dans laquelle est définie la clôture ou celle de l'appelant ? Et si l'appelant n'est pas un

10. Dont la documentation du système de type est disponible ici : https://stw.gitbooks.io/the-encore-programming-language/content/kappa.html

acteur et ne possède pas de promesse courante ? Afin d'éviter tout comportement étonnant, nous interdisons les appels à `forward` dans les clôtures. Il n'y a cependant actuellement aucune vérification dans la bibliothèque. Une telle vérification devrait être possible utilisant la résolution de noms sur `forward`, mais elle n'est faite que localement, au niveau des `let`.

3.6 Génération de code

Nous concluons cette partie en donnant un exemple d'acteur généré par notre bibliothèque.

Nous présentons figure figure 11 un acteur accompagné de son équivalent dé-sucré, légèrement simplifié afin d'en faciliter la lecture. Chaque méthode donne 2 méthodes dans le code généré. Par exemple, la méthode `$meth_set` contient le code initialement contenu dans la méthode `get`, et la nouvelle méthode `get` contient le mécanisme d'appel asynchrone, avec notamment la création de promesse. Le champ mutable `x : int` a été remplacé par `$val_x_281 : int Domain.DLS.t`. Les lectures de `x` deviennent un appel à `DLS.get` (ligne 18) et les écritures à `DLS.set` (ligne 7).

```
let a = object%actor
  val mutable x = 0
  method set n =
    x <- n
  method get = x
end
```

```
let a =
 object (self)
  val $val_x_281 = DLS.new_key (fun _ -> 0)
  method ($meth_set) n =
   DLS.set $val_x_281 n
  method set n =
    let (p, fill) = Promise.create () in
    Actor.send self
    (Multiroundrobin.process fill @@ fun _ ->
     self#$meth_set n);
    p
  method ($meth_get) =
   DLS.get $val_x_281
  method get =
   let (p, fill) = Promise.create () in
   Actor.send self
   (Multiroundrobin.process fill @@
    fun _ -> self#$meth_get);
   p
 end
```

FIG. 11 – Acteur dé-sucré

4 BENCHMARKS

Nous étudions ici les performances de la bibliothèque et des différentes optimisations présentées. Les sources de ces benchmarks sont disponibles ici : actors-ocaml/bench. Ces benchmarks ont été réalisés sur une machine possédant un CPU Intel i7 1165G7 (8 threads) et 8 Go de RAM.

4.1 Explications

Nous comparons différentes implémentations des algorithmes suivants :

Ackermann : Un acteur unique calcule la fonction d'Ackermann, qui n'est pas récursive terminale.

Syracuse : Il s'agit encore d'une fonction récursive, mais la récursion est terminale, ce qui devrait permettre d'utiliser forward pour n'allouer qu'une seule promesse pour tout le calcul.

Ring : Un anneau d'acteurs calcule le modulo de manière naïve. Nous comparons ici les différents appels en cas d'acteurs multiples.

Produit de matrices : Il permet de mesurer les performances de notre bibliothèque pour des applications calculatoires. Nous comparerons nos résultats avec une implémentation utilisant les fonctions « bas niveau » de la bibliothèque standard d'OCaml.

4.2 Acteur unique et chaines de promesses

Le but de ces benchmarks est d'évaluer le surcoût introduit par l'ordonnancement, le passage de messages et les allocations de promesses en cas d'acteur unique.

— Dans les versions « Await », chaque appel de méthode crée une promesse, et chaque promesse est attendue avec `Promise.await`.

— Les versions « Eio » utilisent les promesses de la bibliothèque « Eio ». Les appels de méthodes asynchrones ont été remplacés par des `Fiber.fork_promise`.

— « Forward » remplace les appels de méthodes terminaux par des appels `forward`. Dans le cas d'Ackermann, il reste un appel non terminal.

— « Sync » remplace tous les appels par des appels synchrones.

	Rate	Await	Eio	Forward	Sync	Native
Await	$267\,s^{-1}$	—	−21%	−23%	−96%	−100%
Eio	$338\,s^{-1}$	27%	—	−3%	−95%	−100%
Forward	$348\,s^{-1}$	30%	3%	—	−95%	−100%
Sync	$6447\,s^{-1}$	2314%	1808%	1751%	—	−92%

Table 1 – Ackermann(3, 4)

	Rate	Eio	Await	Forward	Sync	Native
Eio	$1753\,s^{-1}$	—	−2%	−55%	−96%	−99%
Await	$1785\,s^{-1}$	2%	—	−54%	−96%	−99%
Forward	$3879\,s^{-1}$	121%	117%	—	−92%	−99%
Sync	$46\,838\,s^{-1}$	2572%	2524%	1108%	—	−84%

Table 2 – Syracuse

Les promesses de la bibliothèque Eio sont généralement plus rapides que notre version « Await », qui n'utilise qu'on sous-ensemble des fonctionnalités disponibles. `Forward` offre comme attendu de meilleures performances, en particulier dans le cas de Syracuse, où la récursion terminale permet de ne créer qu'une unique promesse pour tout le calcul.

Enfin, l'appel synchrone améliore drastiquement les performances dans ce cas, car nous évitons le surcoût du passage de message et de la création de promesses.

	Rate	Sync	Await	Forward
Sync	$4.15\,s^{-1}$	—	−3%	−65%
Await	$4.28\,s^{-1}$	3%	—	−64%
Forward	$11.8\,s^{-1}$	185%	176%	—

TABLE 3 – Ring

4.3 Acteurs multiples, appels terminaux

Nous étudions désormais l'écart de performances avec des acteurs multiples, utilisant différents domaines.

Dans ce cas, l'appel synchrone n'offre aucun gain de vitesse, car les acteurs étant exécutés dans des domaines différents, l'appel synchrone est traduit en `send; get`. La proximité entre « Await » est « Sync » n'est donc pas surprenante. Encore une fois, « Forward » réduit le nombre de promesses et accélère notablement le calcul.

4.4 Applications calculatoires

Comparons désormais une implémentation « acteurs » du produit de matrice avec une version parallèle plus classique réalisée avec la bibliothèque `domainslib`. Chaque ligne est calculée en parallèle des autres, il y a donc n acteurs de créés pour une matrice de taille n.

	Rate	Nothing	Actor	Domainslib
Seq	$3.22\,s^{-1}$	—	−64%	−68%
Actor	$8.90\,s^{-1}$	176%	—	−11%
Domainslib	$10.0\,s^{-1}$	211%	12%	—

TABLE 4 – Produit de matrices 512×512

Les deux implémentations parallèles sont plus rapides que la version séquentielle (Seq). La version « Domainslib » est plus rapide que la version acteurs, mais Domainslib utilise un accès direct aux threads, exposant les programmeurs à une gestion entière et explicite du parallélisme, en plus des problèmes de data races, deadlocks etc. Notre bibliothèque garantit l'absence de data races et n'utilise pas de verrous. Malgrès cela, elle n'est que 11% plus lente ! Ce résultat est prometteur pour un prototype, nous pourrions encore améliorer les performances en perfectionnant l'ordonnanceur, en réduisant le nombre d'appels de méthodes effectués pour envoyer un message à un acteur, ou en réduisant le nombre d'indirections sur les promesses.

5 CONCLUSION

Nous réalisé avec succès une bibliothèque d'acteurs pour le langage OCaml. Certains points clairement identifiés restent à améliorer, comme la détection d'une utilisation de `forward` dans une clôture, ou la traduction automatique vers des appels optimisés. Mais le prototype réalisé tient ses promesses de *thread-safety* tout en ayant un surcoût acceptable. Il est évident que nous aurions eu plus de liberté si nous avions écrit un compilateur de toutes pièces, mais l'intérêt d'une bibliothèque est qu'elle s'intègre dans un langage ayant fait ses preuves, et bénéficie donc du compilateur, du système de types et des optimisations du langage hôte, en plus d'être plus accessible au grand public. L'entièreté du code est disponible sur https://github.com/Marsupilami1/actors-ocaml.

RÉFÉRENCES

Gul Agha. 1986. An Overview of Actor Languages. In *Proceedings of the 1986 SIGPLAN Workshop on Object-Oriented Programming* (Yorktown Heights, New York, USA) *(OOPWORK '86)*. Association for Computing Machinery, New York, NY, USA, 58–67. https://doi.org/10.1145/323779.323743

Kiko Fernandez-Reyes, Dave Clarke, Elias Castegren, and Huu-Phuc Vo. EasyChair, 2018. Forward to a Promising Future. EasyChair Preprint no. 113. https://doi.org/10.29007/nm5t

Robert H. Halstead. 1985. MULTILISP : a language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.* 7 (1985), 501–538.