Synchronisation des *futures*, implémentations et conséquences

Amaury MAILLÉ

14 avril 2019

Table des matières

Listings

Liste des tableaux

1 Introduction

En programmation, un *future* [4] est une case vide associée à un calcul : lorsque ce calcul produit une valeur, la case qui lui est associée est remplie avec cette même valeur. Dotés d'opérations permettant d'attendre que la case soit remplie, les *futures* apparaissent naturellement en programmation asynchrone. La possibilité d'attendre qu'un calcul asynchrone produise une valeur permet au programmeur de placer des points de synchronisation dans son programme. Ainsi une tâche A pourra utiliser un *future* pour attendre qu'une autre tâche B produise une valeur, que la tâche A pourra ensuite exploiter. Ces points de synchronisation sont plus facilement maniables par le programmeur que les primitives classiques de programmation concurrente, comme les *mutex* ou les *threads*. Le code écrit ressemble plus à un code séquentiel, et le programmeur n'a plus à se soucier lui-même de la synchronisation, qui sera effectuée automatiquement, généralement grâce à une unique instruction.

Les *futures* sont utilisés aussi bien dans des langages fonctionnels (MultiLisp [11]), que dans des langages orientés objets (C++, Java), ou encore dans les langages acteurs ou à objets actifs [1], c'est à dire des langages similaires aux langages objets, ou les entités (appelées acteurs) communiquent entre elles de façon asynchrone en échangeant des messages. Encore [6], ABS [13], ASP [7], le framework Akka [17] sont des langages soit à acteurs, soit à objets actifs. Nous nous focaliserons ici sur les langages à objets actifs en raison de l'activité de recherche actuellement présente dans ce domaine, tout en nous permettant de regarder un langage plus "classique", à savoir le langage JavaScript, pour son intéressante construction `Promise`.

Nous allons présenter ici différentes constructions possibles autour des *futures*, en observant les problèmes qu'elles permettent de résoudre, comment est-ce qu'elles sont spécifiées, implémentées, et quelle garanties est-ce qu'elles apportent.

Dans un premier temps, nous commencerons par présenter plus en détail ce que sont les *futures*, les langages à objets actifs et les mécanismes de synchronisation des *futures* en section 2. Dans un second temps, nous présenterons quatre constructions basées sur les *futures*, d'abord sur un langage orienté objet (JavaScript), puis sur des langages à objets actifs, en section 3. Enfin, nous comparerons ces différentes constructions selon plusieurs axes, afin de synthétiser leurs avantages et inconvénients, ainsi que les raisons qui pourraient pousser à leur utilisation, en section 4.

2 *Futures* et langages à objets actifs

2.1 *Futures* et *promises*

Un *future* représente le résultat d'un calcul asynchrone, et peut être vu comme une case vide qui, à terme, contiendra une valeur. Un *future* est dit en attente (*pending*) lorsqu'il est vide, et résolu (*resolved*) quand une valeur a été placée à l'intérieur. Un *future* est une entité de synchronisation : une tâche peut se synchroniser avec la résolution d'un *future*, c'est à dire attendre, passivement, que le *future* soit résolu pour poursuivre ses traitements. Un *future* est dit "sur `T`" si la valeur qui le résout est de type `T`.

En programmation asynchrone, la création d'une tâche asynchrone renvoie immédiatement un *future* qui, à terme, contiendra le résultat de cette tâche. En C++, on trouve par exemple la fonction `std::async` qui permet de créer une tâche asynchrone. `std::async` permet d'invoquer une fonction de façon asynchrone, et renvoie immédiatement un *future* qui contiendra, à terme, le résultat de la fonction invoquée. Dans le cas de C++, l'aspect asynchrone se traduit par la création d'un nouveau *thread*, mais ce n'est pas nécessaire le cas dans toutes les implémentations des *futures*. Par exemple, dans les langages à objets actifs que nous étudierons plus loin, un *thread* supplémentaire n'est pas nécessairement créé lors d'un appel asynchrone.

Nous présentons par la suite deux classifications différentes des *futures*, les outils nécessaires pour les manipuler, et le concept, assez proche, de *promise*.

2.1.1 *Futures* implicites et *futures* explicites

On distingue traditionnellement deux catégories de *futures* : les *futures* implicites et les *futures* explicites [5]. On peut appliquer cette catégorisation sur trois aspects différents : la création des *futures*, l'accès aux *futures* et le typage des *futures*.

Listing 1 – Création implicite de *future* en ProActive

```
1 Foo f = PAActiveObject.newActive(Foo.class, args);
2 Bar b = f.foo();
3 System.out.println(b.bar());
```

Création Un *future* est explicite par création s'il existe une construction spécifique pour le créer. Par exemple, dans MultiLisp la construction `(future X)` lance l'évaluation parallèle de l'expression X et renvoie un *future* ; on demande explicitement à créer un *future*. Un *future* est implicite par création si rien dans le code n'indique que l'on vient de créer un *future*. Par exemple, dans la librairie Java ProActive [3], appeler une méthode sur un objet obtenu via ProActive renvoie un objet en apparence non *future*, mais au comportement de *future*. Par exemple, dans le *listing* 1, l'instruction `b.bar()` va déclencher une synchronisation : `b` se comporte comme un *future*, et l'appel à `bar` ne sera exécuté qu'une fois que `foo` aura terminé son exécution.

Accès Un *future* sur `T` est explicite d'accès s'il ne peut pas être manipulé de façon transparente comme un objet de type `T` et possède des opérations propres. En C++, les *futures* sont explicites d'accès : la classe `future` présente les opérations `get`, `valid`... De plus, si l'on considère une clase `Foo` dotée d'une méthode `f`, on ne peut pas écrire `async(create_foo, args).f()` : un *future* sur `Foo` ne possède que des opérations de `future`, et non les opérations de `Foo`. Un *future* est implicite d'accès s'il peut être manipulé de la même façon que le type qu'il contient. Par exemple dans Multilisp, on peut écrire `(+ (future X) (future Y))` ; en supposant que X et Y s'évaluent en des valeurs numériques, + pourra travailler avec les *futures* comme s'il travaillait avec des nombres : on est donc sur un accès implicite.

Typage Un *future* est explicite par typage si le type statique d'une expression permet toujours de savoir si l'expression est un *future*. Un *future* est implicite par typage autrement. Par exemple, en C++, un *future* sur un type `T` est noté `std::future<T>`. Le typage du langage permet de savoir, en lisant le code, si une variable est de type *future* ou non : les *futures* sont explicitement typés. A l'inverse, dans Java avec ProActive, si une fonction prend en paramètre un `Foo`, il est impossible de savoir, grâce au type statique, si ce `Foo` est un *future* ou non : les *futures* sont implicites par typage dans ProActive. [1]

2.1.2 *Synchronisation control-flow* et *synchronisation dataflow*

Henrio [12] propose une autre classification des *futures*, cette fois basée sur le mécanisme de synchronisation utilisé. Deux mécanismes sont identifiés : une synchronisation dite *control-flow*, et une synchronisation dite *dataflow*.

Dans une synchronisation *control-flow*, la synchronisation suit le flot de contrôle du programme : quand le programmeur demande à récupérer la valeur contenue dans le future, il obtiendra précisément la valeur renvoyée par la tâche asynchrone, même si cette valeur est elle-même un future. Dans ce cas-là, le programmeur devra demander à récupérer la valeur de ce nouveau, et ainsi de suite, jusqu'à obtenir une valeur exploitable. En raison de leur typage explicite fort, les *futures* de C++ présentent une synchronisation *control-flow*. Considérons les *futures* de C++ dans le *listing* 2.

`u64` est en entier non signé de 64 bits, `compute` est une fonction longue à s'exécuter, `future<T>` est le type *future* sur T et `future<T>::get` est l'opération, bloquante, permettant de récupérer la valeur contenue dans un *future*. On se rend compte que le *future* `f1` sera rempli avec le *future* renvoyé à la ligne 4 par l'invocation d'`async`, et que les *futures* `f10` et `f20` seront remplis avec les valeurs calculées par `compute`. Chaque *future* est rempli par un `return` bien défini, et il n'est pas possible, en un seul appel de méthode, d'obtenir la valeur finale depuis le `future<future<u64>>` : il est nécessaire d'effectuer `f1.get().get()` pour extraire la valeur de la factorielle. Dans le cas de `f20`, qui appelle directement `fact`, un seul `get` est nécessaire.

Dans une synchronisation *dataflow*, la synchronisation est traduite par la mise à disposition d'une valeur exploitable. Nous avons vu que dans une synchronisation *control-flow*, si la valeur produite par la

1. On ne peut donc pas dire que les *futures* sont nécessairement explicites (resp. implicites) par typage si le langage est typé statiquement (resp. dynamiquement)

Listing 2 – *Futures control-flow* en C++

```
u64 compute(int);

future<u64> async_compute(int n) {
    return async(compute, n);
}

int main() {
    future<future<u64>> f1 = async(async_compute, 10);
    future<u64> f20 = async(compute, 20);
    future<u64> f10 = f1.get();
    printf("%llu\n", f10.get() + f20.get());
    return 0;
}
```

Listing 3 – *Futures dataflow* en MultiLisp

```
(defun compute (n)
    ; Long running function
)

(defun async_compute (n)
    (future compute n)
)

(setq res (+ (future compute 20) (future async_compute 10)))
```

tâche est un *future*, alors ce sera ce *future* qui sera placé dans le *future* associé à la tâche. Dans une synchronisation *dataflow*, la synchronisation ne s'effectuerait pas à ce moment-là. La tâche dépendant de ce nouveau *future* serait exécutée, toujours de façon asynchrone, jusqu'à atteindre son `return`. Si la valeur produite n'est pas un *future*, le *future* initial est rempli avec cette valeur. Si la valeur produite est un nouveau *future*, la nouvelle tâche est lancée toujours de façon asynchrone, et on répète le processus jusqu'à atteindre une tâche qui produise une valeur non *future*. On parle également d'attente *wait-by-necessity*. Les *futures* de MultiLisp présentent cette synchronisation. Considérons le code précédent exprimé dans le langage MultiLisp, dans le *listing* 3, à titre d'exemple.

L'instruction `future X` lance l'évaluation de l'expression `X` en parallèle du flot d'exécution qui a déclenché l'instruction.

Nous pouvons observer deux choses dans ce code, une à l'écriture, une à l'exécution :

— A l'écriture, il n'est plus nécessaire d'effectuer des `get` pour pouvoir extraire les valeurs des *futures*. Nous sommes sur des *futures* implicites d'accès, on peut donc manipuler le résultat de `future X` de la même manière que le résultat de `X` ; la capacité d'utiliser le retour de `(future X)` est propre à MultiLisp, et n'est pas lié à la synchronisation *dataflow* ;

— A l'exécution, on peut constater qu'il y deux attentes, mais en un seul point. `async_compute` renvoie `future compute n`, et est appelée de façon asynchrone, ce qui crée un *future* de *future*. En C++ il était nécessaire d'effectuer deux `get` pour obtenir une valeur exploitable. Ici, le simple fait d'utiliser le *future* obtenu par appel asynchrone à `async_compute` déclenche la synchronisation. Le programme attend jusqu'à ce qu'une valeur exploitable soit produite. On peut imaginer cela comme un `get` qui attendrait récursivement que tous les *futures* sous-jacents soit résolus.

Ces deux exemples ne sont pas représentatifs de l'utilisation des *futures*. Ils sont uniquement à but d'illustration.

Listing 4 – Chaînage dans Encore

```
1  import Task
2  active class Producer
3    def produce() : int
4      42 + 1
5    end
6  end
7  active class Main
8    def main() : unit
9      let
10       p = new Producer
11       l = fun (x : int)
12             println(x)
13             x + 1
14           end
15     in
16       println(get(p ! produce() ~~> l))
17     end
18   end
19 end
```

2.1.3 Outils de synchronisation des *futures*

Dans le cas des *futures* explicites d'accès il est nécessaire de travailler directement sur le *future* pour effectuer la synchronisation. Nous allons nous intéresser à deux outils : l'instruction `get` et le principe du chaînage.

get L'instruction `get` est l'outil le plus fréquent pour manipuler les `futures`. Lorsqu'on utilise `get` le *thread* appelant est bloqué jusqu'à ce que le *future* soit résolu, et `get` renvoie alors la valeur qui a résolu le *future* (si le *future* est déjà résolu, `get` retourne immédiatement la valeur qui l'a résolu). La fonction `get` existe dans des langages comme Encore en tant qu'instruction, ou C++ en tant que membre de la classe `future`. L'appellation peut varier d'un langage à l'autre ; par exemple, dans Python, on trouve la méthode `result` membre de la classe `Future`, qui correspond précisément au comportement de `get`.

chain Le chaînage (instruction `chain` ou `then` selon les langages) est utilisé pour attacher du code qui sera exécuté une fois un *future* résolu. Ce code se comportera comme une fonction implicite unaire. Cette fonction implicite sera invoquée de façon asynchrone au moment où le *future* sera résolu, et recevra en paramètre la valeur qui a servi à résoudre le *future*. En raison du caractère asynchrone de cette invocation, `chain` va renvoyer un *future*, qui contiendra le résultat de l'exécution de la fonction implicite. On peut ainsi chaîner des *futures* les uns aux autres. Dans Python, la fonction `add_done_callback` permet de s'approcher du chaînage en attachant une fonction à la résolution d'un *future*, bien que l'absence d'invocation asynchrone ne permette pas de vrai chaînage. Dans le langage Encore [6], l'opérateur $\rightsquigarrow$ fait office de construction `chain`. Dans JavaScript, ce sont les opérations `then` et `catch` qui le permettent [2]. Considérons le *listing* 4 en langage Encore, à titre d'exemple simple [3].

Dans le listing 4, l'opérateur `!` dénote une invocation asynchrone, l'opérateur $\rightsquigarrow$ dénote le chaînage, `get` est l'instruction de synchronisation permettant d'extraire la valeur d'un *future*. Le flot d'exécution est le suivant : la méthode `produce` de l'objet référé par `p` est invoquée de façon asynchrone, ce qui produit un *future*. La résolution de ce *future* est liée avec l'exécution de `l`, une fonction unaire ; le résultat de ce chaînage est un *future*, sur lequel est appelée `get`.

Lorsque `produce` s'achève, elle produit la valeur 43, ce qui résout le *future* renvoyé par `p!produce()` avec la valeur 43. La fonction `l` est ensuite appelée avec 43 en paramètre. `l` affiche 43, puis renvoie 44, ce

2. Standard ECMA6
3. Ce programme fait partie de la suite de test d'`encorec`, le compilateur du langage Encore. En ligne, consulté le 7/04/2019

Listing 5 – Comparaison des *futures* et *promises* en C++

```
1  #include <future>
2
3  int f_future() {
4      // Some long running computation that produces a value "v" of type int
5      return v;
6  }
7
8  void f_promise(std::promise<int>& p) {
9      // Some long running computation that produces a value "v" of type int
10     p.set_value(v);
11 }
12
13 int main() {
14     // Futures are produced through the "async" function
15     std::future<int> f1 = std::async(f_future);
16
17     // Promises are created manually. Asynchronous invocation is delegated to
18     // the programmer. Here we spawn a thread.
19     std::promise<int> p;
20     std::thread t(f_promise, std::ref(p));
21
22     // When set_value is called on "p" with a value "v", "f2" is resolved with
23     // that same value "p".
24     std::future<int> f2 = p.get_future();
25
26     printf("f1 = %d, f2 = %d\n", f1.get(), f2.get());
27     return 0;
28 }
```

qui résout le *future* renvoyé par le chaînage. L'opération `get` débloque et extrait la valeur 44 du *future* obtenu par chaînage, et le `println` de la ligne 16 est exécuté, et affiche 44 à l'écran.

2.1.4 *Promises*

Les *promises* sont un concept très lié aux *futures* et offrent une alternative plus explicite aux *futures* : les *futures* sont créés par des opérations comme `async`, ou l'invocation de méthodes sur des objets actifs, et sont automatiquement résolus quand la tâche produit une valeur ; une *promise* est explicitement créée par le programmeur, et est explicitement remplie par le programmeur quand une valeur est disponible. En contrepartie, les *promises* ne proposent aucune des garanties des *futures* : un *future* est nécessairement rempli lorsque la tâche associée produit une valeur, alors qu'il n'y a aucune garantie qu'une *promise* sera bien remplie une et une seule fois. Le *listing* 5 illustre les différences entre *promises* et *futures* en C++.

Les fonctions `f_future` et `f_promise` effectuent le même traitement qui produit une valeur de type `int` stockée dans une variable `v`. `f_future` renvoie cette valeur telle quelle, `f_promise` remplit une *promise* passée en paramètre avec cette valeur `v`, grâce à la méthode `set_value`. Nous pouvons observer les principales différences entre *promises* et *futures* :

- `f_future` n'utilise pas de syntaxe spéciale, son intégration avec le concept de *future* est transparente ; `f_promise` travaille explicitement sur une *promise* `p` passée en paramètre, et remplit cette *promise* avec la méthode `set_value` ;
- `f_future` est invoquée au travers de la méthode `async` pour que l'appel soit asynchrone, ce qui produit un *future* en retour ; `f_promise` est invoquée dans un *thread* créé explicitement par le programmeur, et reçoit en paramètre une *promise* explicitement créée par le paramètre ;
- Le *future* associé à l'invocation asynchrone de `f_future` est obtenu par l'appel à `async` ; le *future* "associé" [4] à l'invocation de `f_promise` est obtenu par la méthode `get_future` sur la *promise* `p`.

4. Il s'agit ici d'un abus de langage. Le *future* est associé à la *promise* `p`, et non à l'invocation de `f_promise`

2.2 Acteurs et objets actifs

Un langage acteur est un langage dans lequel le programmeur manipule des entités *monothreadées* et indépendantes les unes des autres, appelées acteurs. Les acteurs s'exécutent en parallèle les une avec les autres. Ils communiquent entre eux par passage de messages asynchrones. Un langage à objets actifs est un langage acteur, où les acteurs sont des objets, et dans lequel la communication entre objets se fait au travers d'appels de méthodes asynchrones. En termes techniques, un acteur possède une file de messages en attente, et les traite un par un de façon séquentielle.

Les langages acteurs sont utilisés dans la conception systèmes distribués, ainsi que dans la conception de systèmes multiprocessus locaux. L'aspect *monothreadé* des acteurs permet de garantir une absence de *data race* au sein d'un acteur, ce qui simplifie la programmation : il n'y a plus besoin de réfléchir en termes de *mutex* ou autres primitives de synchronisation.

2.2.1 Utilisation des *futures* dans les langages acteurs

L'aspect asynchrone des communications entre objets actifs fait intervenir les *futures* pour la transmission des résultats. Lorsqu'un objet actif A appelle de façon asynchrone une méthode sur un objet actif B, A n'a aucune garantie que B exécutera la méthode immédiatement ; l'appel sera traduit par l'ajout d'un message dans la file de l'objet actif B. Compte tenu du caractère *monothreadé* des acteurs, attendre la réponse de B pourrait bloquer A pour une très longue durée. Pour éviter cela, l'appel asynchrone à la méthode de B va immédiatement renvoyer un *future* qui contiendra le résultat une fois que B aura fini de traiter l'appel. A va donc conserver ce *future* jusqu'à en avoir besoin, et à ce moment-là A pourra commencer son attente. Cela permet également à A de lancer d'autres appels asynchrones, avant de commencer son attente sur les *futures*.

2.2.2 Manipulation des *futures* dans les langages acteurs

Les langages acteurs supportent, pour la plupart, les constructions usuelles sur les *futures* (`get` et `chain`). Certains d'entre eux sont dotés d'une construction supplémentaire, `await`. Cette construction est similaire à `get`, mais, si le *future* n'est pas encore résolu, le traitement du message actuel sera suspendu et l'objet actif commencera à traiter un autre message. Lorsque le moment de poursuivre le traitement du message mis en pause arrivera, `await` sera à nouveau exécuté, pour vérifier si le *future* est désormais résolu. S'il l'est, le traitement du message se poursuivra ; sinon, le traitement sera à nouveau suspendu et l'objet actif traitera un autre message pendant ce temps.

3 Une étude de constructions récentes basées sur les *futures*

Dans cette section, nous présentons quatre constructions basées sur ou exploitant les *futures*. Les *Promises* du langage JavaScript, les constructions `ParT` et `forward` du langage à objets actifs Encore, et le concept de *Dataflow explicit futures*, qui s'intéresse à la synchronisation des *futures* explicites. Pour chacun de ces concepts, nous proposerons un problème lié aux *futures*, les outils existants pour le résoudre, en quoi ce concept répond à ce problème et en quoi il pourrait, éventuellement, être amélioré.

3.1 Les *Promises* de JavaScript

Le langage JavaScript est un langage de programmation orienté objet, basé sur le paradigme de programmation évènementielle. Le langage JavaScript est implémenté dans la plupart des navigateurs Web en raison de son intégration native avec l'API DOM, qui lui permet de manipuler des éléments des pages Web, afin d'y ajouter un aspect dynamique. Ce dynamisme est basé sur l'interaction avec l'utilisateur : JavaScript permet de récupérer des évènements en réaction aux actions de l'utilisateur. Ainsi, quand l'utilisateur clique sur un bouton, il est possible de réagir à cet évènement en JavaScript.

Pour réagir aux évènements, JavaScript utilise le principe de *callbacks* ou *event handlers*, qui sont des fonctions déclenchées en réaction à un évènement. Par exemple, pour réagir au clic sur un bouton, on pourrait écrire `button.addEventHandler("click", function() { console.log("Button clicked"); })`. Cette réaction écrit "Button clicked" dans la console de *debug* du navigateur chaque fois que l'utilisateur clique sur le bouton auquel la réaction est attachée (on dit que l'évènement est émis). La fonction

`addEventHandler` permet d'attacher une réaction (la fonction anonyme) à un évènement (ici "click") sur un objet (ici, `button`). Cette approche présente cependant trois inconvénients :

— Il est possible d'attacher une réaction après qu'un évènement ait été émis ; dans ce cas, l'évènement est perdu et il n'est plus possible d'y réagir. L'un des exemples les fréquents est l'évènement "DOMContentLoaded" qui est émis une et une seule fois : lorsque la page Web a fini de charger. Si le code JavaScript qui enregistre une réaction est émis après que la page ait fini de charger, la réaction ne sera jamais déclenchée, car l'évènement ne sera jamais réémis ;

— L'utilisation de *callbacks* peut conduire à la création d'un *callback hell* : des *callbacks* vont enregistrer d'autres *callbacks* qui vont eux-mêmes enregistrer d'autres *callbacks* et ainsi de suite. Le flot d'exécution devient donc complexe à comprendre et à superviser : à un instant t il est difficile de déterminer de façon certaine comment est-ce que le programme réagira à un évènement ;

— Le langage JavaScript ne permet pas de spécifier des liens de dépendances entre *callbacks*. Par exemple, si un *callback* A attache un *callback* B à l'exécution d'un évènement, A et B n'ont pas connaissance l'un de l'autre [5]. Par conséquent, exprimer l'idée "Si l'exécution de A échoue, détacher B de l'évènement" est très complexe, voire impossible.

La problématique est donc la suivante : proposer un outil qui puisse permettre de réagir à *posteriori* à un évènement, tout en permettant d'enchaîner des réactions, et en permettant de contrôler facilement l'ajout et la suppresion de réactions. Le concept des *promises* a été intégré au langage JavaScript, sous la forme de la classe `Promise` afin de répondre à ce problème. Les `Promises` de JavaScript présentent un comportement de *future* et de *promise* en même temps. Ine `Promise` JavaScript est une case vide, tout comme les *futures*, mais qui est créée explicitement et remplie explicitement, comme les *promises* classiques. De plus, le programmeur ne possède pas de méthode équivalente à `get` ou `await` pour obtenir le résultat de la `Promise` : il doit nécessairement passer par du chaînage. Nous nous appuyons sur Madsen et al. [15] pour présenter le concept de `Promise`, et en évaluer les implications et limites.

3.1.1 Présentation

Une `Promise` est créée autour d'une fonction binaire, dont les paramètres sont symboliquement nommés `reject` et `resolve`. Une `Promise` peut être dans deux états différents : en attente (*pending*), ou résolue (*fulfilled*). On considère que la `Promise` est en attente tant qu'elle n'est pas remplie, et remplie lorsqu'elle est résolue ou rejetée. Une `Promise` est résolue avec une valeur `v` lorsque la fonction binaire qu'elle gère appelle son paramètre `resolve` avec la valeur `v` ; une `Promise` est rejetée avec une valeur `v'` lorsque la fonction binaire qu'elle gère appelle son paramètre `reject` avec la valeur `v'`, ou lève une exception. Nous illustrons ce comportement dans le *listing* 6.

Listing 6 – Création de `Promise` en JavaScript

```
1 // Create a new promise, resolved with value 42
2 new Promise(function(resolve, reject) { resolve(42); });
3
4 // Create a new promise, rejected with value -1
5 new Promise(function(resolve, reject) { reject(-1); });
6
7 // Create a new promise, rejected with value "Exception raised"
8 new Promise(function(resolve, reject) { throw "Exception␣raised"; });
```

Le code est assez explicite, la ligne 2 déclare une promesse résolue avec la valeur 42, la ligne 5 déclare une promesse rejetée avec la valeur -1 et la ligne 8 déclare une promesse rejetée car la fonction qu'elle gère lève une exception. Dans chaque exemple, `resolve` est la fonction permettant de résoudre la promesse avec une valeur, et `reject` est la fonction permettant de la rejetter avec une valeur.

Contrairement aux *futures*, les `Promise` ne possèdent pas de méthode native permettant d'obtenir les valeurs qui ont servi à les résoudre ou à les rejeter. A la place, les `Promises` proposent deux opérations permettant d'effectuer un chaînage afin de travailler sur la valeur de remplissage : `then` et `catch`. Nous les illustrons dans le *listing* 7 avant de les expliquer.

5. A moins que le programmeur ne prenne lui-même des mesures pour enregistrer cette association dans une structure de données qu'il peut manipuler

Listing 7 – Fonctions `then` et `catch` en JavaScript

```
1  // Chain on a resolved future
2  new Promise(function(resolve, _) {
3      resolve(42);
4  }).then(function(resolveValue) {
5      console.log(resolveValue); // 42
6  });
7
8  // Chain on a rejected future, using then
9  new Promise(function(_, reject) {
10     reject(-1);
11 }).then(function(_) { }, function(rejectValue) { // Note that then takes two
       arguments here
12     console.log(rejectValue); // -1
13 });
14
15 // Chain on a rejected future, using catch
16 new Promise(function(_, reject) {
17     reject(-1);
18 }).catch(function(rejectValue) {
19     console.log(rejectValue); // -1
20 });
```

`then` La fonction `then` est une fonction binaire (on peut omettre le second paramètre), dont les paramètres sont deux fonctions symboliquement nommées `onResolve` (resp. `onReject`). Ces fonctions reçoivent en paramètre la valeur qui a été utilisée pour résoudre (resp. rejeter) la `Promise`, et permettent de réagir par rapport à cette valeur. Ainsi, dans le *listing* 7, le `then` de la ligne 4 permettra l'exécution du `console.log` lorsque la `Promise` de la ligne 2 sera résolue. De façon similaire, le `then` de la ligne 11 permettra l'exécution du `console.log` lorsque la `Promise` de la ligne 9 sera rejetée. [6]

`catch` La fonction `catch` est une fonction unaire dont le paramètre est lui-même une fonction unaire, symboliquement nommée `onReject`. Cette fonction unaire reçoit en paramètre la valeur qui a été utilisée pour rejeter la `Promise`. Ainsi, dans le *listing* 7, le `catch` de la ligne 18 sera exécuté lorsque la *promise* de la ligne 16 sera rejetée [7].

`then` et `catch` permettent de mettre en place du chaînage. A l'image du chaînage des *futures*, ce chaînage de `Promises` créera des nouvelles `Promises` qui seront remplies par l'exécution d'un `return` dans `onReject` ou dans `onResolve`. Il est ensuite possible d'appeler `then` ou `catch` sur la nouvelle `Promise` pour chaîner de nouvelles réactions. Nous illustrons l'enchaînement de tous ces concepts dans le *listing* 8

La promesse `p1` créée à la ligne 3 est résolue avec la valeur 42 via l'invocation de `resolve` à la ligne 4. L'instruction `then` attache une réaction à la résolution de `p1` : afficher "p1 resolved with value = " suivi de la valeur de résolution (ici, 42). `then` renvoie une `Promise` dépendante `p2` qui sera résolue avec la valeur renvoyée par la fonction de réaction. On peut donc chaîner pour récupérer cette nouvelle valeur, ce qui est fait à la ligne 12. Une fois `p2` résolue, le programme écrit "p2 resolved with value = 84" dans la console.

La promesse `p3` est rejetée avec la valeur -1 à la ligne `19`. On enregistre une réaction à ce rejet via `then` à la ligne 22. Lors de la réaction, une exception est levée : la promesse dépendante `p4` sera rejetée avec pour valeur de rejet une exception. Comme nous l'avons vu, on peut utiliser `catch` pour réagir à un rejet, et la réaction enregistrée à la ligne `27` écrira "p4 rejected with value = Exception occured" dans la console.

`then` à la ligne 12 et `catch` à la ligne 27 renvoient des promesses dépendantes (ignorée pour `then`, `p5` pour `catch`). Étant donné que les fonctions de réaction ne renvoient pas de valeurs, ces deux promesses

6. Lorsque `then` est utilisée pour réagir exclusivement à une résolution, on peut omettre le second paramètre. Lorsqu'on souhaite réagir à un rejet, il est nécessaire de donner les deux paramètres

7. On peut donc gérer le rejet de deux façons : avec `catch`, ou avec `then`

Listing 8 – Utilisation des `Promise` en JavaScript

```
1  // Resolution
2
3  let p1 = new Promise(function(resolve, reject) {
4      resolve(42);
5  });
6
7  let p2 = p1.then(function(resolveValue) {
8      console.log("p1 resolved with value = " + resolveValue); // 42
9      return resolveValue * 2;
10 });
11
12 p2.then(function(resolveValue) {
13    console.log("p2 resolved with value = " + resolveValue); // 84
14 });
15
16 // Rejet
17
18 let p3 = new Promise(function(resolve, reject) {
19     reject(-1);
20 });
21
22 let p4 = p3.then(function(resolveValue) {}, function(rejectValue) {
23     console.log("p3 rejected with value = " + rejectValue); // -1
24     throw "Exception occured !";
25 });
26
27 let p5 = p4.catch(function(rejectValue) {
28     console.log("p4 rejected with value = " + rejectValue); // Exception "
           Exception occured !"
29 });
```

dépendantes seront résolues avec `undefined`.

3.1.2 Synchronisation des `Promises`

Les `Promises` de JavaScript présentent un comportement hybride entre les *futures* et les *promises*. Tout comme les *futures*, il est possible d'effectuer du chaînage, pour créer des `Promises` dépendantes. Tout comme les *promises*, il est possible de déléguer le remplissage à une autre fonction, voire à une autre tâche, en passant les fonctions `resolve` et `reject` en paramètres d'autres fonctions. Nous pouvons donc nous interroger sur le concept de synchronisation pour ces `Promises`. Considérons le *listing* 9.

Listing 9 – Synchronisation des `Promises`

```
1 new Promise(function(resolve, reject) {
2     resolve(new Promise(function(resolve2, reject2) {
3         reject2(42);
4     }));
5 }).then(function(onResolve) {
6     console.log(onResolve);
7 }, function(onReject) {
8     console.log(onReject);
9 });
```

Intuitivement, on peut se dire que le `onResolve` de la ligne 5 est un objet de type `Promise`. En réalité, ce `onResolve` est la valeur 42. Lorsqu'une `Promise` est résolue ou rejetée avec une autre `Promise` [8], c'est la résolution ou le rejet de la seconde `Promise` qui va déterminer si la `Promise` englobante est résolue ou rejetée, et avec quelle valeur. Il est ainsi possible d'enchaîner récursivement plusieurs `Promise` pour obtenir une unique valeur de résolution ou de rejet. Cette propriété s'étend aux fonctions passées en paramètres de `then` et `catch`. Si l'on reprend le *listing* 8, et que l'on change la ligne 9 par une expression de la forme `return new Promise(...)`, le `then` de la ligne 12 sera exécuté lorsque la nouvelle `Promise` sera résolue avec une valeur `v`, et le `resolveValue` de la ligne 12 vaudra précisément `v`.

Nous sommes ici face à une synchronisation *dataflow* : si l'on considère que `then` est équivalent à une sorte de `get`, alors un `then` va récursivement attendre un nombre indéterminé de `Promise`, en une seule instruction.

3.1.3 Implications et limitations

En premier lieu nous pouvons constater que les `Promises` répondent bien à la problématique présentée en introduction : comment exécuter du code asynchrone sans créer un *callback hell*, tout en ayant la possibilité de faire remonter les erreurs à travers la chaîne de *callbacks*, et en ayant la possibilité de réagir même une fois un évènement émis. Les `Promise`, avec les fonctions `then` et `catch` font apparaître un flot de contrôle sémantiquement séquentiel qui met en avant le concept de chaînage (*then*). Les valeurs de résolution, de rejet, ainsi que les exceptions sont propagées dans cette chaîne ; il est en particulier possible de placer un unique `catch` sur toute une chaîne pour pouvoir réagir à n'importe quelle erreur survenue plus tôt. La sémantique proposée par Madsen et al. met par ailleurs en évidence qu'une promesse résolue peut encore déclencher des réactions attachées à posteriori, ce qui permet de résoudre le problème des évènements perdus.

Les `Promises` présentent cependant des limitations et ne sont pas exemptes de problèmes. Madsen et al. [15] ont effectué un recensement de diverses questions posées sur le forum *StackOverflow* en rapport avec les `Promises` de JavaScript, et ont constaté qu'une très grande partie des problèmes soulevés étaient liés au chaînage. En particulier, ils présentent un exemple où la fonction passée en paramètre à `then` renvoie une valeur dans certains cas, mais pas dans d'autres, en raison d'un `return` mal placé ; dans certains cas, la `Promise` dépendante est résolue avec `undefined`. Ces travaux mettent en évidence des difficultés à manipuler les chaînes de `Promises` lorsque les liens entre `Promises` deviennent de plus en plus complexes. En formalisant le concept de `Promise`, Madsen et al. proposent l'idée d'un outil de *debug*, fonctionnant par analyse statique, et permettant la construction d'un graphe représentant les dépendances entre les `Promise`, ainsi que les instructions chargées de les résoudre ; ce graphe permet ainsi de détecter les chaînes de `Promise` brisées ou incorrectes. Madsen et al. proposent un outil permettant la

8. C'est-à-dire que `resolve` ou `reject` est appelée avec une `Promise` en paramètre

création de ce graphe par analyse dynamique dans [2], qui, selon leur conclusion, est "capable de construire le graphe de promesses pour des applications complexes avec un surcoût d'exécution acceptable".

Il est également important de noter que l'absence de préemption et de *multithreading* en JavaScript impacte la façon d'écrire les `Promise`. L'implémentation de `Promise` reflète un aspect important qui est la délégation du chaînage. Lorsqu'une `Promise` est résolue ou rejetée, et que des `Promise` dépendantes sont présentes, celles-ci ne sont pas résolues ou rejetées immédiatement. Madsen et al. rappellent la présence d'une boucle de traitement, qui itère sur les `Promise`, créées par le programmeur ou résultant d'un appel à `then` ou à `catch`. Or, le temps de traitement est très variable d'une `Promise` à une autre ; chaque `Promise` exécute, sans interruption, la fonction passée au constructeur de `Promise` ; pendant ce temps, aucune autre `Promise` ne peut s'exécuter. Cela pousse le programmeur à concevoir son programme en termes de fonctions qui s'exécutent rapidement, afin de ne pas bloquer tout l'aspect asynchrone des `Promise`.

On peut par ailleurs comparer les *promises* au module Python *asyncio*[9], en particulier en ce qui concerne les possibilités offertes au programmeur sur le traitement des calculs asynchrones. Dans *asyncio*, le programmeur peut choisir quand est-ce qu'il bascule en asynchrone, et gérer plusieurs boucles évènementielles en même temps. En JavaScript, la gestion des `Promise` commence une fois que les scripts utilisateurs ont terminé leur exécution, et le programmeur délègue tout le traitement au moteur JavaScript. Cette absence d'accès à la boucle de gestion des `Promise` ne permet par exemple pas au programmeur d'annuler une `Promise`, là où la méthode `cancel` sur les `Task` et `Handle` d'*asyncio* le permet. Il est possible pour le programmeur d'augmenter les `Promise` avec de nouvelles fonctionnalités en réimplémentant une partie du concept, mais cela reste toujours sujet à erreur.

3.2 Délégation de résolution de *futures*

Encore [6] est un langage de programmation orienté objet et à objets actifs, inspiré du langage ABS. Nous commençons par présenter succinctement le langage ABS et le langage Encore, puis nous présenterons la construction *forward*.

3.2.1 ABS et Encore

ABS est un langage de programmation orienté objet concurrent de modélisation. La concurrence d'ABS est inspirée du modèle acteur et de l'ordonnancement coopératif. Les objets d'ABS sont organisés en groupes, appelés groupes d'objets concurrents (GOC). Contrairement au modèle acteur classique où chaque acteur possède son propre fil d'exécution, les objets d'ABS au sein d'un même GOC partagent un fil d'exécution commun. Ainsi, les objets ne possèdent plus une file de messages chacun ; les objets d'un GOC partagent une même file, et c'est un ordonnanceur qui décide quel objet traitera un nouveau message. Les appels de fonctions en ABS peuvent être synchrone (`operator .`) ou asynchrone (`operator !`). Effectuer un appel asynchrone renvoie un *future* sur le type du retour de la fonction appelée. En ABS, le type *future* sur `T` est noté `Fut<T>`.

Pour manipuler les *futures*, ABS propose deux constructions : `future.get` et `await future?`. `future.get` fonctionne comme nous l'avions présenté en section 2.1.3 : si le *future* est résolu, `future.get` renvoie immédiatement la valeur contenue dans le *future*, sinon il bloque jusqu'à ce que le *future* soit résolu. `await future?` laisse l'exécution continuer si le *future* est résolu, sinon l'objet rend la main à l'ordonnanceur pour laisser un autre objet traiter un message.

Encore propose des *futures* explicites par typage, création et accès, avec synchronisation *control-flow*. L'opérateur `!` permet d'invoquer une méthode de façon asynchrone sur un objet actif, à l'image d'ABS. Le type *future* sur `T` est noté `Fut [T]`. Encore présente également les opérations `get` et `await`, qui respectent les principes que nous avons présentés en sections 2.1.3 et 2.2.2. Enfin, comme nous l'avons mentionné précédemment, Encore propose un opérateur de chaînage, l'opérateur $\rightsquigarrow$, qui prend en paramètres un *future* `f` et une fonction implicite unaire `F`. Quand `f` sera résolu, `F` sera exécutée en recevant en paramètre la valeur de résolution de `f`. L'opérateur $\rightsquigarrow$ renvoie un *future* qui, à terme, contiendra la valeur renvoyée par l'exécution de la fonction `F`.

Les *futures* du langage Encore sont explicites, par typage, par création et par accès. Si `T` est un type, `Fut [T]` est un type distinct. Cette distinction est récursive : pour un type `T`, `Fut [T]` est un type distinct

9. `https://docs.python.org/3/library/asyncio.html`

Listing 10 – Implémentation naïve d'un *Broker* dans Encore

```
1 active class Broker
2   val workers: Buffered[Worker]
3   var current: uint
4   def run(job: Job): int
5     val worker = this.workers[++this.current % workers.size()]
6     val future : Fut[int] = worker!start(job)
7     return get(future)
8    end
9 end
```

de `T`, et `Fut [Fut [T]]` est un type distinct des deux précédents. Les *futures* d'Encore présentent une synchronisation *control-flow* : il est nécessaire, lorsque N niveaux de *futures* sont enchaînées, d'effectuer N synchronisations, en l'occurrence à l'aide d'une primitive telle que `get` ou `await`.

3.2.2 Le problème du *broker*

Considérons le problème suivant, classique en programmation, le problème du *broker* : une classe, que nous nommerons `Broker`, reçoit des requêtes, et délègue le traitement de ces requêtes à des classes `Worker`. De façon générale, il est nécessaire que la classe `Broker` présente du parallélisme : nous ne voulons pas que les requêtes soient traitées l'une après l'autre (à moins que des dépendances n'existent entre plusieurs requêtes), nous voulons profiter des possibilités offertes par les machines multi-cœurs modernes. Nous utiliserons les *futures* pour l'aspect asynchrone et profiter de la simplicité d'écriture qu'ils offrent.

Considérons les différentes possibilités d'implémentation du problème du *broker* basées sur les *futures* en Encore. Nous reprenons ici les exemples proposés par Fernandez-Reyes et al. dans [9].

L'implémentation la plus immédiate de la classe `Broker` est proposé dans le *listing* 10. Cette implémentation est assez peu efficace : la méthode `run` va immédiatement bloquer en attendant que le *future* soit résolu. Dans un langage sans objets actifs, plusieurs appels à `run` pourraient s'accumuler en parallèle, mais dans un langage à objets actifs comme Encore, où les acteurs sont *monothreadés*, il est impossible d'effectuer un second appel à la méthode `run` tant que le premier n'est pas terminé. Le *thread* du Broker va donc exécuter les `run` les uns après les autres, sans le moindre parallélisme. Il est donc nécessaire d'optimiser cette solution.

Plusieurs solutions sont envisageables : la méthode `run` peut renvoyer le *future* créé par l'appel au *worker* ; dans ce cas, l'appelant recevra un `Fut[Fut[int]]` [10], et devra effectuer deux synchronisations. Le parallélisme augmente, mais le coût en mémoire est doublé, de même que le travail du programmeur. Le *broker* pourrait utiliser *await* sur le *future* renvoyé par le *worker* ; cette solution a toutefois l'inconvénient d'être très coûteuse en temps et en mémoire, en raison du surcoût induit par la sauvegarde du contexte d'exécution. Sur un système traitant des milliers de requêtes, les performances seraient fortement dégradées. Enfin, le *broker* et le *worker* pourraient travailler sur une *promise*, ce qui résoudrait les problèmes de coût en temps, mémoire et le problème de prolifération de *futures*, mais perd les garanties des *futures*, à savoir la garantie d'obtenir une valeur, et impose au programmeur de travailler explicitement sur des *promises*, ce qui limite la réutilisation du code.

La solution idéale serait de permettre à la méthode `run` de déléguer la résolution du *future* au `Worker`. En d'autres termes, il faudrait que ce soit une autre fonction, elle aussi invoquée de façon asynchrone et donc liée à un *future*, qui remplisse le premier *future*. Nous appelons ce concept la délégation. Fernandez-Reyes et al.[9] implémentent ce concept dans le langage Encore au travers de la construction `forward`.

3.2.3 `forward`

`forward` est une construction particulière qui permet de déléguer la résolution d'un *future* à une autre fonction asynchrone. Nous introduisons le concept de *current future*. Pour une méthode `M` d'un objet `o`,

10. Si l'on est dans un langage à objet actifs. Dans un langage comme Java, cette solution serait acceptable ; en C++, l'impossibilité de copier les *futures* rendrait cette solution non optimale et demanderait plus de travail au programmeur

Listing 11 – Illustration du fonctionnement de `forward`

```
1  active class Broker
2    def run(job: Job): int
3      val worker = this.workers[++this.current % workers.size()]
4      val g = worker!start(job)
5      forward(g)
6
7  active class Main
8    def main(): unit
9      val brk = new Broker()
10     val f = brk.run(new Job(\() -> return 42)
11     // f se resout en 42
```

invoquée de façon asynchrone, le *current future* de `M` est le *future* qui sera résolu par l'exécution de `M`.

`forward` prend en paramètre un *future*, que nous nommerons *future* de résolution. Lorsque `forward` est invoqué avec un *future* `g`, l'exécution de la méthode courante est terminée, et, lorsque `g` sera résolu, le *current future* de la méthode ayant appelé `forward` sera résolu avec la même valeur. Considérons le *listing* 11, qui présente une implémentation du problème du *broker* en Encore, avec `forward`.

Lorsque `run` appelle `forward`, `run` s'arrête (si du code venait après, il ne serait pas exécuté). `worker !process(request)` renvoie un *future*, stocké `g`. `forward g` va lier la résolution du *future* stocké dans `g` à celle du *current future*, renvoyé par `brk.run(...)` dans `main`, et actuellement stocké dans `f` dans `main`. Si la valeur de `f` ne change pas, on peut dire que la résolution du *future* stocké dans `g` déclenchera la résolution du *future* stocké dans `f`.

En termes techniques, `forward` est un chaînage particulier, où la fonction implicite est la fonction identité et où le *future* résultant du chaînage est le *current future* de la fonction invoquant `forward`.

3.2.4 Formalisation et implémentation

Les programmes écrits en Encore sont compilés vers du C, liés avec l'environnement d'exécution du langage Pony [8], puis vers du code machine. Pony est un langage de programmation acteur et orienté-objet, dont l'environnement d'exécution, *libponyrt*[11], est écrit en C. A ce niveau-là, les *futures* d'Encore sont traités de la même manière que les *promises* : ils sont passés en paramètres des fonctions qui doivent les résoudre, et résolus "manuellement". La formalisation de `forward` est effectuée en trois temps : une formalisation d'un langage minimaliste similaire à Encore doté de *futures*, et sans effets de bord ; une formalisation d'un langage minimaliste similaire au premier, mais où les *futures* sont remplacés par des *promises*, similaire au comportement de Pony ; et une formalisation d'une stratégie de compilation du langages à *futures* vers le langages à *promises*.

Nous ne présenterons pas ces trois formalismes en détail, nous nous conterons de les survoler. Nous renvoyons le lecteur curieux vers les travaux de Fernandez-Reyes et al. [9] pour plus de détails.

Futures Dans le langage à *futures*, l'implémentation de `forward` se traduit par un chaînage. Dans ce chaînage la fonction implicite à exécuter une fois le *future* résolu est la fonction identité. Dans l'expression `forward g`, si `f` est le *current future*, `forward` chaîne `f` sur `g` et attache l'identité en fonction de chaînage. Quand `g` est résolu, `f` reçoit la valeur de résolution. L'utilisation du chaînage permet de garantir que `f` sera bien résolu ; la garantie que `g` sera bien résolue vient de l'utilisation des *futures* et de leur formalisation. [12]

Promises L'implémentation de `forward` dans un langage possédant exclusivement des *promises* demande de repenser la construction. `forward` équivaut à créer une tâche asynchrone, effectuer un chaînage sur le *future* obtenu suite au lancement de cette tâche, et à résoudre un *future*. Le formalisme proposé par Fernandez-Reyes et al. ne comporte pas la construction `forward`, mais contient les opérations de création de tâches et de chaînage. A la différence du langage à *futures*, les *promises* ne sont pas résolues automatiquement, mais manuellement *via* l'instruction `fulfil`. De même, les *promises* sont créées manuellement

11. GitHub Pony
12. `forward` est donc un chaînage particulier. Là où `f` $\rightsquigarrow$ `e` crée un *future*, le chaînage de `forward` ne produit pas de nouveau *future*, mais travaille sur un *future* préexistant

là où les *futures* sont créés automatiquement. A l'image du `get` des *futures*, `get` appliqué à une *promise* permet d'en extraire la valeur de résolution. Tous les composants nécessaires pour implémenter `forward` sont présents. Le processus de compilation des *futures* vers les *promises* peut donc se faire. Pour préserver la sémantique de `forward`, la compilation doit apporter la garantie que les *promises* seront bien résolues.

Compilation Considérons que l'on dispose d'une opération *fulfil*, qui prend en paramètre une *promise* `p` et une valeur `v`. `fulfil(p, v)` résout `p` avec la valeur `v`. Nous pouvons à présent définir la compilation pour le lancement de tâche et le chaînage. Pour garantir que le lancement d'une tâche résout bien la *promise* qui lui est associée, on peut compiler le code de la tâche en `fulfil(p, e)`, où `p` est la *current promise* de la tâche, et `e` l'expression qu'elle doit exécuter. De même, une compilation de la continuation d'un chaînage garantissant une résolution de la *promise* associée est `fulfil(p, e)`, avec `p` la *current promise* de la continuation, et `e` l'expression qu'elle doit exécuter.

3.2.5 Implications et limitations

Tout d'abord, nous pouvons observer que `forward` répond précisément à la problématique proposée : comment déléguer la résolution d'un *fuuture* à une autre fonction asynchrone ? `forward` évite les constructions `Fut[Fut[T]]` grâce à l'utilisation du chaînage, et a pour seul coût mémoriel le suivi de l'état des *futures*, des chaînes et des tâches, qui est assez faible. De plus, la formalisation de `forward`, effectuée aussi bien dans un langage similaire à Encore, que dans un langage similaire à celui utilisé dans l'environnement d'exécution d'Encore, prouve les propriétés que nous cherchions : non imbrication des *futures*, garantie de résolution du *future* délégué, et empreinte mémoire faible.

Par ailleurs, Fernandez-Reyes et al.[9] montrent également les résultats de l'utilisation de `forward` sur le problème du *broker* dans le langage Encore. Dans ce contexte, les performances montrées par `forward` sont très importantes, que ce soit par rapport à l'utilisation de `get` seul, ou à l'utilisation de `await` et de `get`. L'empreinte mémoire est également très réduite, en particulier lorsque l'on fait la comparaison avec `await`, qui a le surcoût de sauvegarde de la pile pour permettre un changement de contexte. Nous citons ici un résultat obtenu sur le programme *broker* que nous avons mentionné en introduction, sur une charge de 10000 tâches

- En utilisant exclusivement `get`, le *broker* consomme 12 kB de mémoire et écoule la charge en 183 secondes ;
- En utilisant `get` et `await` simultanément, le *broker* consomme 48 kB de mémoire et écoule la charge en 92 secondes ;
- En utilisant `forward`, le *broker* consomme 8.5 kB de mémoire et écoule la charge en 4 secondes.

Cependant, la formalisation proposée par Fernandez-Reyes et al. suppose que les langages utilisés dans la formalisation sont exempts d'effets de bords. La formalisation n'est donc prouvée valide que pour des langages sans effets de bords. Dans l'implémentation de `forward` dans Encore, on constate que certaines garanties ne sont plus respectées, bien que l'implémentation fonctionne en pratique. Par exemple, rien n'empêche un utilisateur de créer un cycle de *futures* à l'aide de l'implémentation de `forward` dans Encore, ce qui n'est pas possible dans le formalisme.

3.3 Combinateurs parallèles pour *pipelines* spéculatifs

Dans la plupart des langages orientés objets il est possible d'exprimer le parallélisme de façon assez simple, à l'aide d'outils comme les tâches, les *threads*, les *futures*... En revanche, il est plus délicat d'exprimer de la coordination sur le parallélisme. Par exemple, comment faire pour lancer en parallèle plusieurs tâches, et exprimer "Je souhaite récupérer le résultat de la première des tâches qui termine et interrompre les autres" ? Un langage comme JavaScript présente la construction `Promise.race` qui prend en paramètre un ensemble de `Promise` et renvoie la première `Promise` résolue ou rejetée, mais ne permet pas d'interrompre les autres `Promise` de l'ensemble.

3.3.1 ParT

Pour répondre à ces problèmes, Fernandez-Reyes et al. proposent *ParT*, "une abstraction parallèle asynchrone pour les calculs en pipeline spéculatif"[10], par la suite implémentée dans le langage Encore. Nous nous appuierons ici exclusivement sur la spécification formelle.

Dans les sections à venir, la notation *ParT* réfère au concept, la notation *Par* τ réfère à une instance de *Par* travaillant sur un type τ. La notation *Fut* τ représente un *future* sur le type τ.

Un *ParT* peut être vu comme une représentation d'un ensemble de calculs parallèles. On peut appliquer diverses opérations sur un *ParT*, au travers d'opérations appelées "combinateurs". Par exemple, on peut appliquer une fonction de façon asynchrone à l'ensemble des futurs résultats des différents calculs (voir opérateur $\gg$) ; une telle opération renvoie un nouveau *ParT*, contenant des *futures* qui seront résolus avec les résultats de l'application de la fonction. L'idée d'utiliser des combinateurs est issue du langage Orc[14], un langage conçu pour tester et expérimenter les combinateurs, et a ensuite été reprise par McCain au cours de sa thèse de master[16] dans laquelle il formule un concept nommé *Par* τ, puis par Fernandez-Reyes et al. dans [10], où ils formalisent *Par* τ. Nous présenterons les principaux combinateurs en section 3.3.2.

Un *ParT* peut être construit de trois façons. L'expression {} construit un *ParT* vide, qui ne possède aucune valeur. L'expression $\{-\}$ construit un *Par* τ à partir d'une valeur de type τ. L'expression `a || b`, avec `a` et `b` des *Par* τ construit un *Par* τ, qui indique la possibilité d'un parallélisme entre `a` et `b`. Deux constructeurs spéciaux, appelés constructeurs de *lift*, que Fernandez-Reyes et al. notent respectivement † et $^\circ$ permettent de créer un *Par* τ, respectivement à partir d'un *Fut* τ et d'un *Fut* (*Par* τ).

En termes plus techniques, et c'est le point de vue que nous adopterons dans la suite, un *Par* τ peut être vu comme un ensemble contenant quatre catégories de valeurs : ϵ, ou l'absence de valeur, représentée par {} (on parle de *Par* τ vide) ; une valeur `v` de type τ, représentée par {`v`} ; une valeur de type *Fut* τ ; et une valeur de type *Fut* (*Par* τ).

3.3.2 Combinateurs principaux

Nous présentons ici les deux combinateurs principaux de *ParT* : $\gg$, et $\gg=$.

Combinateur de séquencement ($\gg$) Le combinateur $\gg$ permet de créer un *pipeline* parallèle. $\gg$ prend en paramètre un *Par* τ `p` et une fonction `f` de τ vers τ'. Le résultat de $\gg$ est un *Par* τ' dont les éléments sont des *futures*, représentant l'application asynchrone de `f` sur les éléments de `p`. $\gg$ n'est donc pas bloquant. Une synchronisation ne se produira que lorsque l'utilisateur voudra accéder aux éléments contenus dans le *Par* τ'.

Combinateur *bind* ($\gg=$) Le combinateur $\gg=$ prend en paramètres un *Par* τ, et une fonction de τ vers *Par* τ'. $\gg=$ applique la fonction à chaque élément du *Par* τ de façon asynchrone et renvoie un *Par* τ' contenant les *futures* résultant de l'application asynchrone de `f` sur les éléments. Tout comme $\gg$, l'opérateur de *bind* n'est pas bloquant, et aucune synchronisation n'est effectuée.

3.3.3 Spéculation

Le parallélisme spéculatif de *ParT* est mis en place au travers du combinateur $\ll$, qui permet de récupérer et traiter le premier résultat non vide d'un ensemble de calculs parallèles, ensemble représenté par un *ParT*. Le concept de résultat vide intervient lorsqu'un *Par* τ contient des *Fut* (*Par* τ). Nous traiterons de ce cas en détail.

Fernandez-Reyes et al. présentent les sémantiques de *ParT* dans [10]. Nous nous appuierons sur cette sémantique par la suite, d'un point de vue haut niveau. Nous commencerons par présenter l'objectif des combinateurs `peek` et *prune* ($\ll$), puis nous présenterons leur fonctionnement. Enfin, le *listing* 12 illustrera le fonctionnement de *ParT*, au niveau de ses combinateurs principaux et de la spéculation.

Combinateur spéculatif interne (`peek`) Le combinateur `peek` est utilisé exclusivement en interne par l'implémentation, et permet de mettre en place le parallélisme spéculatif. `peek` prend en paramètre un *Par* τ et renvoie un *Maybe* τ, où *Maybe* τ représente un type optionnel, de la même façon qu'en Haskell. Ce *Maybe* τ vaudra `Nothing` si le *Par* τ était vide, ou si le *Par* τ ne contenait que des *Fut* (*Par* τ) qui se sont tous résolus en *Par* τ vide. Autrement, le *Maybe* τ vaut `Just v`, où `v` est la première valeur exploitable trouvée dans le *Par* τ. Une valeur exploitable est soit une valeur de type τ, soit la valeur d'un *Fut* τ résolu. Considérons un *Par* `int` qui contient l'ensemble de trois valeurs { *Fut*, *Fut* `3`, `4` }, avec *Fut* un *future* non résolu et *Fut* `X` un *future* résolu avec la valeur `X`. `peek`, appelé sur ce *Par*,

renverra `Just 3`, car le premier élément exploitable qu'il rencontre est un *future* résolu avec la valeur `3`. Si les valeurs étaient dans cet ordre : { *Fut*, 4, *Fut* `3` }, `peek` renverrait `Just 4`.

Combinateur spéculatif *prune* (≪) Le combinateur ≪ prend en paramètres une fonction `f` et un *Par* τ, et déclenche la spéculation. ≪ appelle, de façon synchrone, la fonction `f` en lui passant en paramètre un *Fut* (*Maybe* τ). Ce *future* sera résolu avec `Nothing` si le *Par* τ est vide, ou ne contient que des *Fut* (*Par* τ) qui se résolvent tous en des *Par* τ vides. Autrement le *future* sera résolu avec `Just v`, où `v` est la première valeur exploitable contenue dans le *Par* τ.

Nous allons à présent nous intéresser au fonctionnement de ces combinateurs. Nous reprenons le concept de *current future*, tel que définit dans la section 3.2.3. Par abus de langage nous considérerons que l'expression "*Par* τ vide" signifie "*Par* τ ne contenant aucun élément, ou dont tous les éléments sont des *Fut* (*Par* τ) résolus en *Par* τ vide".

Fonctionnement de `peek` Le combinateur `peek` est technique. L'idée de `peek` est de parcourir les éléments d'un *Par* τ une seule fois, et de renvoyer la première valeur exploitable qu'il trouve, ou `Nothing` si le *Par* τ est vide. Un problème majeur se pose ici : comment `peek` peut-elle produire une valeur si le *Par* τ ne contient que des *futures* non résolus ? Afin de conserver l'intérêt de la spéculation, `peek` ne peut pas effectuer un `get` ou un `await` sur les *futures* : la spéculation doit retourner une valeur au plus tôt. Si *Par* τ ne contient aucun élément, ou contient une valeur non *future*, un `return` semble être la solution immédiate. Mais si le *Par* τ ne contient que des *futures* non résolus, que renvoyer une fois tous les éléments du *Par* τ parcourus ? La solution proposée par Fernandez-Reyes et al. est la suivante : `peek` est toujours appelée de façon asynchrone, et ne contient aucun `return`. L'appel asynchrone produit un *future*, que nous nommerons `result`. A présent, nous allons étudier comment est-ce que ce *future* est résolu, selon les types d'éléments que `peek` trouve dans le *Par* τ.

Le cas du *Par* τ vide Lorsque `peek` traite un *Par* τ vide, `peek` résout `result` avec `Nothing`, puis lance la phase d'empoisonnement que nous détaillerons dans le prochaine section. L'idée de cette phase est de s'assurer que le *future* `result` soit résolu une et une seule fois.

`peek` sur une valeur de type τ Lorsque `peek` traite une valeur `X` de type τ, `peek` résout `result` avec `Just X`, puis lance la phase d'empoisonnement.

`peek` sur une valeur de type *Fut* τ Lorsque `peek` traite une valeur de type *Fut* τ, deux cas se présentent. Soit le *future* est résolu avec une valeur `X` et `peek` doit résoudre `result` avec `Just X`. Soit le *future* n'est pas encore résolu et `peek` doit exprimer l'idée "Quand ce *future* sera résolu, il faudra qu'il remplisse `result` avec sa valeur". Dans les deux cas, `peek` crée un chaînage similaire à ce qu'on trouve dans `forward` : une fois le *future* résolu avec une valeur `X`, `result` sera résolu avec `peek X` [13].

`peek` sur une valeur de type *Fut* (*Par* τ) Le cas *Fut* (*Par* τ) est plus complexe que le précédent. Un *Fut* τ est garanti de produire une valeur exploitable pour `peek`. Un *Par* τ peut être vide, et donc un *Fut* (*Par* τ) peut se résoudre en *Par* τ vide, qui n'est pas une valeur exploitable pour `peek`. La stratégie de résolution de cette situation est assez complexe, aussi nous n'entrerons pas dans les détails. L'idée principale est d'appeler récursivement `peek` sur le reste du *Par* τ de façon asynchrone, ce qui produit un *future* `h`. `peek` va également attacher un chaînage sur le *Fut* (*Par* τ). Ce chaînage et `h` vont alors concourir pour résoudre `result` selon la sémantique de `peek`.

Fonctionnement de ≪ Lorsqu'une tâche t effectue `f << par`, avec `f` une fonction et `par` un *Par* τ, ≪ invoque `peek` de façon asynchrone et obtient un *future* (le *future* `result`). ≪ appelle ensuite, de façon synchrone, `f(result)`.

13. `X` est transformé en un *Par* τ de la forme `X` puis passé à un nouvel appel à `peek` afin de pouvoir déclencher une phase d'empoisonnement si nécessaire

Listing 12 – Fonctionnement de *ParT*

```
1  active class Main
2      def pruner(f : Fut[Maybe[int]]) : Par[int]
3          var res = get(f)
4          match res with
5              case Just(x) => liftv(x)
6              case Nothing => empty[int]()
7          end
8      end
9
10     def main() : unit
11         var futures = new [Fut[int]](10)
12         for x <- [0..9] do
13             futures(x) = async(running(x))
14         end
15
16         var part = lift_fut_array(futures)
17         var part2 = part >> fun (x : int) => x * 2
18         var part3 = bind(fun (x : int) : Par[int]
19                              println(x)
20                              return liftv(x)
21                          end, part2)
22         (pruner << part3) >> fun (x : int) : int
23                                  println(x)
24                                  return x
25                              end
26     end
27 end
```

La spéculation prend tout son intérêt lorsque le *ParT* contient exclusivement des *futures*. Dans cette situation, la spéculation se traduit par une course à la résolution du *future* de `peek`. Tous les *futures* contenus dans le *ParT* reçoivent un chaînage qui leur fera résoudre le *future* `result`, et le premier *future* qui produira une valeur exploitable écrira cette valeur dans le *future* `result`.

Nous illustrons le fonctionnement des combinateurs et de la spéculation à travers l'exemple en *listing* 12, écrit en Encore [14].

`Fut[T]` dénote le type *future* sur `T`, `Par[T]` dénote le type *Par* sur `T`, `bind` est l'opérateur de *bind* (≫=). Pour un tableau `array`, `array(x)` est l'équivalent de `array[x]` en langage C.

Les lignes 11 à 14 créent un tableau de dix *futures* et le remplissent avec les résultats de dix appels asynchrones à la fonction `running`. `running` est une fonction qui prend en paramètre un `int x`, attend un temps aléatoire de 0 à 10 secondes, puis renvoie `x`. La ligne 16 transforme le tableau de `Fut[int]` en un `Par[int]`, *via* la fonction `lift_fut_array`. La ligne 17 illustre le comportement du combinateur ≫.

De façon parallèle, ≫ va itérer sur les éléments de `part` (qui sont des *futures*) et leur attacher un chaînage qui multiplie par 2 leur valeur de résolution. L'ensemble des *futures* obtenus via ces chaînages est ensuite transformé en `Par[int]`, qui est passé au combinateur `bind`. L'utilisation du chaînage permet de ne pas bloquer le *thread* principal en enregistrant une continuation plutôt qu'en attendant la résolution des *futures*. Le `Par[int]` résultant est stocké dans `part2`.

De façon parallèle, `bind` va appliquer la fonction qui lui est passée en paramètre sur les éléments du `Par[int] part2` (éléments qui sont toujours des `Fut[int]`. La fonction en question affiche la valeur qui a résolu le *future* et renvoie un `Par[int]` contenant cette même valeur. Tout comme ≫, `bind` va utiliser des chaînages pour conserver l'aspect asynchrone. Les `println` seront exécutés de façon asynchrone une fois les *futures* résolus.

La ligne 22 réalise la spéculation *via* l'opérateur ≪, et les lignes 22 à 24 affichent le résultat de la spéculation *via* l'opérateur ≫. Nous nous concentrons sur la spéculation.

14. Le *pruning* n'est pas implémenté dans Encore, ce code ne compile actuellement pas

$\ll$ commence par appeler `peek(part3)` de façon asynchrone et obtient un *future* `f` sur le résultat de `peek`. $\ll$ invoque ensuite de façon synchrone `pruner(f)`. Intéressons nous au comportement de `peek`. Nous avons l'avantage de n'avoir que des *futures* dans le *ParT*. Le comportement de `peek`, quel que soit le *ParT* est de parcourir tous ses éléments et de prendre une décision par élément. Dans le cas de *futures* sur des types autres que *Par*, soit le *future* est résolu, et `peek` s'arrête immédiatement et résout son *current future* avec la valeur du *future* qu'il a trouvé ; soit le *future* est en attente, et dans ce cas il faut déléguer à plus tard la résolution du *current future* de `peek`, avec un chaînage.

Étudions ce qu'il se passe si `running(1)` n'attend pas du tout et renvoie 1 immédiatement. L'appel à `running(1)` était asynchrone et a renvoyé un *future* `f1`. Ce *future* a été placé dans le *ParT* `part` à la ligne 18. Nous avons ensuite appelé le combinateur $\gg$ sur `part`. Ce combinateur a attaché un chaînage à chaque *future* de `part`. Nommons `f2` le *future* résultant du chaînage sur `f1`. `f2` est placé avec les autres *futures* résultant des chaînages dans le `Par[int] part2`. Enfin, nous avons appelé `bind` sur `part2`, ce qui a mis en place de nouveaux chaînages. Nommons `f3` le *future* résultant du chaînage sur `f2`. C'est ce *future* sur lequel `peek` travaille. Nous avons donc la chaîne `f1` $\rightarrow$ `f2` $\rightarrow$ `f3`. `running(1)` résout `f1` avec 1, ce qui résout `f2` avec 2, ce qui résout `f3` avec 2 (et affiche 2). Le *future* passé à `pruner` se résout donc en `Just (2)`.

Si `running(9)` est le plus rapide de tous les appels, alors `peek` va attacher des chaînage à tous les autres *futures*, et renvoyer la valeur qui a résolu le *future* en bout de chaîne associée à `running(9)`.

Si nous injections une valeur, par exemple `3` en plein milieu du *ParT*, et qu'aucun des *futures* le précédent n'est résolu, `peek` renverrait `Just 3`.

Nous passons volontairement sous silence le mécanisme d'empoisonnement (que nous verrons dans la prochaine section), afin de ne pas rendre l'explication plus technique. Ce mécanisme garantit que les chaînages inutiles, une fois le *current future* de `peek` résolu, seront bien détruits et n'écraserons pas la valeur de résolution du *current future*.

3.3.4 Empoisonnement et destruction

Le fonctionnement de `peek` présente un problème majeur : supposons que l'on ait un *Par* `Int` de cette forme : *Fut* `Int`, *Fut* (*Par* τ), $\{\}$, 3 . `peek` va déclencher du chaînage pour les deux premiers éléments, puis, arrivé à 3, immédiatement renvoyer `Just 3`. Le *future* `result` (celui que `peek` doit résoudre) contiendra donc `Just 3`, mais les chaînages déclenchés précédemment vont tenter à leur tour de remplir ce *future*. Pour parer à cette situation, `peek` procède à un empoisonnement des tâches et chaînages.

L'environnement d'exécution de *ParT* contient l'ensemble des tâches et chaînages en cours, chacun doté d'un indicateur d'empoisonnement. Lorsque `peek` produit une valeur, `peek` passe l'indicateur d'empoisonnement du *future* `result` à "actif". S'ensuit alors une phase de propagation. L'environnement rassemble les tâches et chaînages dépendants du *future* empoisonné, et les empoisonne à leur tour, récursivement.

A mesure que le poison se propage dans les tâches et chaînes du *ParT*, une phase de terminaison, ou destruction, est mise en place. Cette phase a pour objectif de détruire les tâches et chaînes qui ne sont plus nécessaires dans le calcul. Nous n'entrerons pas dans le détail de comment est-ce que la nécessité d'une chaîne ou d'une tâche est déterminée. Le procédé est référé à un *tracing garbage collector*.

3.3.5 Propriétés

L'abstraction *ParT* présente deux propriétés formelles : une absence de *deadlock* et une garantie de sûreté des tâches. La garantie de sûreté des tâches représente l'idée qu'un *future* marqué comme non nécessaire durant la phase d'empoisonnement ne peut pas devenir nécessaire par la suite. Par extension, la décision de détruire une tâche ou une chaîne ne peut pas provoquer d'erreurs : il n'est pas possible qu'une tâche ou chaîne ait soudainement besoin d'une tâche / chaîne détruite. L'absence de *deadlock* représente l'idée qu'un *pipeline* ne peut pas bloquer. Fernandez-Reyes et al. prouvent que la construction *ParT* ne permet pas la création de dépendances cycliques entre *futures*[15], et prouvent également qu'il

15. Cette preuve est effectuée sur un langage fonctionnel sans effets de bord. Tout comme dans `forward`, on peut obtenir des cycles entre *futures* dans l'implémentation de *ParT* dans Encore

n'est pas possible de se retrouver avec un ensemble de tâches, *futures* et chaînes où il n'est plus possible de progresser. L'utilisation d'un *ParT* est donc sûre.

3.4 Une synchronisation *dataflow* pour les *futures* explicites

Nous avons présenté précédemment deux classifications des *futures* : explicite / implicite et *control-flow* / *dataflow*. Nous pouvons, intuitivement, observer que l'aspect explicite ou implicite du typage détermine quelle forme de synchronisation. Un *future* explicitement typé sur un type τ possède une méthode `get` qui renvoie, intuitivement, une valeur de type τ ; les *futures* implicitement typés n'exposent pas directement le programmeur au type qu'ils produisent, et on peut donc plus facilement imaginer que `get` ou qu'un accès direct produise une valeur non *future*, peu importe le nombre de *futures* imbriqués. Par ailleurs, on peut intuitivement se rendre compte qu'il n'est pas possible de typer une fonction récursive terminale renvoyant un *future* explicitement typé sans ajouter de synchronisation : chaque niveau de récursion devrait ajouter un niveau de *future*.

Henrio [12] s'est intéressé à la possibilité d'intégrer une synchronisation *dataflow* aux *futures* explicites, dans une construction nommée *Dataflow explicit Future*, ou "DeF". Dans cette section nous considérerons par abus de langage que *future* explicite signifie "explicitement typé", et que *future* implicite signifie "implicitement typé".

3.4.1 Sur une imbrication finie de *futures*

Une synchronisation *dataflow* sur des *futures* explicites n'est pas *totalement* impossible, quoique technique à mettre en œuvre selon le langage. On peut, grâce à la métaprogrammation, proposer une solution partielle, pour les *futures* explicites avec un niveau déterminé de *futures* imbriqués en C++. Nous l'illustrons ici, avec la métaclasse *helper* `future_type_helper`, et la classe `dataflow_future`. Nous ne montrons dans le *listing* 13 qu'une méthode permettant de typer la fonction `get` de la classe `dataflow_future`.

La problématique est la suivante : étant donné un type τ, potentiellement `dataflow_future` sur un type τ', comment typer le retour de la méthode `get` ? Si τ n'est pas un `dataflow_future`, `get` doit renvoyer un τ. Sinon, il existe un type τ' tel que $\tau =$ `dataflow_future<`τ'`>`. Nous appliquons ce processus jusqu'à obtenir un τ' qui ne soit pas un `dataflow_future`. La métaclasse `future_type_helper` permet de traverser une chaîne de `dataflow_future` jusqu'à obtenir le premier type non `dataflow_future`.

La classe `future_type_helper` existe en deux versions : une dans laquelle elle est paramétrée par un `dataflow_future`, et une dans laquelle elle est paramétrée par tout autre type. Dans les deux cas, elle expose un type, nommé `V`. Si le type paramétrique `T` n'est pas un `dataflow_future`, `V` = `T`. Autrement, le type paramétrique est de la forme `dataflow_future<T>` : `V` est donc défini comme le `V` de `future_type_helper` paramétré avec `T`. Cette définition récursive permet de traverser la chaîne de `dataflow_future` jusqu'à atteindre le premier type `T'` qui ne soit pas un `dataflow_future`. En bout de chaîne, `V` est finalement défini comme ce type `T'`.

Nous pouvons donc définir la classe `dataflow_future` pour lui donner un comportement *dataflow* : si le type paramétrique `T` n'est pas un `dataflow_future`, `get` doit renvoyer un `T`. Sinon, `get` doit renvoyer le premier type paramétrique de la chaîne qui n'est pas un `dataflow_future`, et c'est précisément ce type qui est défini par `future_type_helper<T>::V`. On peut ainsi typer `get`. En ce qui concerne la définition de `get` : si on n'est pas sur un *future*, `get` renvoie la valeur stockée dans le *future* ; sinon, `get` effectue un `get` sur le *future* imbriqué.

3.4.2 Sur une imbrication non bornée de *futures*

Cette construction est toutefois insuffisante : il n'est en effet toujours pas possible de typer une fonction récursive terminale. Si on tente de typer une telle fonction, il est nécessaire de passer par un *template* pour pouvoir absorber une quantité indéterminée de *futures* dans le type de retour. En contrepartie le compilateur devra étendre récursivement le *template* jusqu'à épuisement de la mémoire. Pour résoudre ce problème, Henrio propose la construction `Fut«T»`. Cette construction se lit "potentiellement *future*" et représente un *future* avec un nombre indéterminé de *futures* sous-jacents, qui convergent vers une valeur de type `T`. Cette construction est dotée d'une méthode `get`, dont le type de retour est `T`, et qui,

Listing 13 – Synchronisation *dataflow* pour *futures* finis en C++

```
template<typename T>
class dataflow_future;

template<typename T>
struct future_type_helper {
    typedef T V;
};

template<typename T>
struct future_type_helper<dataflow_future<T>> {
    typedef typename future_type_helper<T>::V V;
};

template<typename T>
class dataflow_future<dataflow_future<T>> {
public:
    typename future_type_helper<T>::V get() {
        return value.get();
    }

private:
    dataflow_future<T> value;
};

template<typename T>
class dataflow_future {
public:
    T get() {
        // ... Wait for "value" to actually have a value
        return value;
    }

private:
    T value;
};
```

à l'exécution, attend récursivement sur les *futures* jusqu'à produire une valeur exploitable. Par ailleurs, `Fut«T»` est doté d'une propriété de typage qui permet de substituer une valeur de type `T` à une valeur de type `Fut«T»` : le *future* est immédiatement résolu avec cette valeur. Cela permet de mettre en place une forme de réutilisation de code.

`Fut«T»` apporte donc les avantages de deux mondes : d'une part, on retrouve l'une des garanties fortes de la synchronisation *dataflow*, à savoir la production d'une valeur exploitable en un seul `get` ; d'autre part, l'aspect explicite supprime l'inconvénient que nous avions observé sur MultiLisp : le programmeur possède un contrôle sur la synchronisation, et peut décider de quand est-ce qu'il va la déclencher, tandis que dans MultiLisp une synchronisation pouvait arriver à tout moment. Enfin, la possibilité de convertir une valeur de type `T` en `Fut«T»` permet une réutilisation de code, similaire à ce que nous avions dans MultiLisp ou ProActive. Toutefois, *DeF* ne possède pas, à ce jour, d'implémentation.

4 Une étude des caractéristiques des *futures*

Nous avons présenté en section 2 trois aspects des *futures* : l'aspect explicite ; l'aspect synchronisation (*control-flow* ou *dataflow*) ; et l'aspect manipulation (`get`, `chain`, `await`). En section 3 nous avons présenté plusieurs constructions basées sur les *futures*, qui tirent partie de ces différents aspects. Nous allons ici confronter les composants de chacun de ces aspects, afin d'étudier leurs avantages et inconvénients, et ce qu'ils permettent ou ne permettent pas de faire.

4.1 *Futures* implicites et *futures* explicites

Nous avons précédemment présenté la différence entre *futures* implicites et explicites. Nous avons observé que cette distinction peut s'effectuer sur trois axes : la création, le typage et l'accès.

Création La création explicite de *future* est présente dans des langages comme C++ (`async`), MultiLisp `future`, Encore (`operator !`)... La création explicite de *futures* permet d'identifier précisément quand est-ce que les *futures* sont créés ; c'est le programmeur qui choisit s'il souhaite créer des *futures* ou utiliser un objet non *future*. Cela permet d'écrire du code plus optimisé, comme on a pu le voir avec MultiLisp. La création implicite apparaît par exemple dans ProActive, une bibliothèque permettant de manipuler des objets actifs en Java. Dans ProActive, l'invocation de méthode sur un objet actif crée un *future*, qui sera résolu par le `return` de la méthode invoquée. Cette forme de création implicite de *future* est automatique, et donc ne demande pas au programmeur de mettre en place des mécanismes particuliers, ce qui lui permet d'écrire du code plus simplement.

Typage Les *futures* typés explicitement offrent une sûreté au programmeur : il n'y a pas de risque qu'une opération sur un objet qui n'est pas un *future* déclenche une synchronisation. En revanche, un typage explicite rend impossible l'écriture de fonctions récursives terminales, car il est impossible de typer leur valeur de retour [16]. De plus, il n'est pas possible de transtyper : si une fonction attend un *future* sur τ en paramètre, il n'est pas possible de lui passer une valeur de type τ en paramètre. [17] Les *futures* typés implicitement apportent leurs avantages et inconvénients, qui sont étroitement liés à l'aspect explicite ou implicite de l'accès. Nous les traiterons dans le paragraphe suivant.

Accès Un accès explicite est plus riche en possibilités qu'un accès implicite, car il traite le *future* comme une entité concrète. C'est ainsi qu'on peut créer des constructions comme `forward` ou *ParT*. L'accès explicite permet également le choix entre les opérations `get` et `await`, et offrent un choix pertinent au programmeur en termes de design. L'accès explicite est toutefois dangereux en présence de *futures* implicitement typés, car manipuler un *future* sur τ de la même manière qu'une valeur de type τ résultera en une erreur. L'accès implicite n'est possible que sur des *futures* typés implicitement, et contribue à la réutilisation de code, mais peut aussi être source de *deadlocks* : comme le programmeur ne sait plus qu'est-ce qui est un *future* ou non, des cycles de *futures* peuvent apparaître sans qu'il en ait conscience.

16. DeF le permettrait
17. DeF le permettrait également

Il y a donc huit formes de *futures* différents si l'on considère ces trois axes. Lorsqu'on souhaite utiliser des *futures* le choix de la forme peut être guidé de plusieurs façons. Si le programmeur manipule des *futures* implicites par typage, et que tout fonctionne bien, il n'est pas nécessaire de considérer des *futures* explicites d'accès. Si le programmeur manipule des *futures* implicites d'accès et qu'il rencontre des *deadlocks* complexes, peut-être que des *futures* explicites par accès seraient préférables, pour mieux visualiser les points de synchronisation. Le niveau d'expertise du programmeur peut également entrer en compte. Les *futures* implicites sont en général plus souples à l'utilisation, et donnent un aspect séquentiel au code en abstrayant les détails du parallélisme, au risque de dissimuler des synchronisations. Les *futures* explicites sont moins souples et plus techniques, mais permettent un contrôle beaucoup plus important sur la synchronisation et sur le flot d'exécution.

Une conclusion générale que nous pouvons tirer de ces trois axes de comparaison est le choix entre la souplesse d'écriture et le contrôle offert au programmeur : est-ce qu'on obtient automatiquement des *futures* ou est-ce qu'ils sont créés par un mécanisme spécifique ? Est-ce qu'on souhaite un typage dynamique et souple, avec peu de garanties en sûreté, ou est-ce qu'on préfère avoir un typage extrêmement fort qui garantisse qu'on sache exactement ce qu'on manipule, de façon statique ? Et est-ce qu'on souhaite pouvoir passer des *futures* et des variables classiques indifféremment à des fonctions, au risque de *deadlock*, ou est-ce qu'on est prêt à accepter la distinction entre valeurs *futures* et valeurs non *futures* ?

4.2 Synchronisation *control-flow* et synchronisation *dataflow*

Nous avons précédemment présenté la différence entre les synchronisations *control-flow* et *dataflow*. Nous allons à présent étudier ces deux formes de synchronisation plus en détail.

Control-flow La synchronisation *control-flow* se produit lorsque l'exécution d'une expression, en général un `return`, résout la synchronisation (i.e, le `get` débloque et retourne une valeur). Cette forme de synchronisation présente donc une garantie extrêmement forte : il est possible, pour chaque *future*, de savoir très précisément quel ensemble d'expressions peut le résoudre. Par ailleurs, dans le cas où l'on se retrouve avec plusieurs niveaux de *futures*, il y a une garantie forte que l'exécution de N `get` produira une valeur. Une telle forme de synchronisation est très utile dans un contexte d'ordonnanceur par exemple, où il est plus intéressant de savoir où en est une tâche, plutôt que d'obtenir la garantie qu'une synchronisation produira une valeur.

Dataflow La synchronisation *dataflow* se produit lorsque c'est la mise à disposition d'une donnée qui résout la synchronisation. On a donc une garantie forte que l'évaluation d'un *future* produira une valeur exploitable, contrairement à une synchronisation *control-flow* où l'on doit nécessairement effectuer autant de synchronisation manuelles qu'on a de niveaux de *futures*. Une telle synchronisation est préférable dans la plupart des cas, en raison du confort d'écriture qu'elle apporte. Dans les langages à *futures* implicites d'accès, elle est présente par nécessité, et contribue à l'écriture visuellement séquentielle et dénuée de *futures* du code. Dans les langages à *futures* explicites d'accès, elle apporte le confort de l'écriture d'un unique `get` ou `await` pour obtenir une valeur exploitable. Les *futures* de Python présentent une synchronisation *control-flow*, mais on pourrait facilement les doter d'une synchronisation *dataflow*. Considérons le *listing* 14.

Avec une telle classe, appeler `get_result` sur un `DataflowFuture` produira nécessairement une valeur, peu importe le nombre de `DataflowFuture` sous jacents. C'est donc bien une forme de synchronisation *dataflow*.

***Futures* implicites, *futures* explicites** On peut mettre en évidence un lien fort entre l'aspect implicite / explicite du typage et de l'accès des *futures* et la synchronisation qu'ils présentent. En C++, les *futures* sont explicites à tous les niveaux. Une synchronisation *dataflow* totale apparaît immédiatement comme impossible car on ne peut typer certaines fonctions asynchrones (nous en avons montré une version partielle dans le *listing* 13). Inversement, les *futures* de MultiLisp, implicites par typage et accès, ne peuvent pas présenter de synchronisation *control-flow* : il n'y a ni méthode pour déclencher une synchronisation, ni de distinction entre les *futures* et les non *futures*. Dans le cas où le typage est

Listing 14 – Synchronisation *dataflow* sur *futures* explicites d'accès et implicites par typage

```
class DataflowFuture:
    # Attend que le future soit resolu
    def await_result(self):
        pass

    def get_result(self):
        self.await_result()
        if type(self._value) == type(self):
            return self._value.get_result()
        else:
            return self._value
```

Listing 15 – *Deadlock* par utilisation d'un `ThreadPoolExecutor` en Python

```
def wait_on_future(executor):
    # Add a second function. It won't run the lambda until we return.
    fut = executor.submit(lambda x, y : x + y, 1, 2)

    # Deadlock : we need to return so the executor can run the lambda. In order
    # for us to return, the executor has to run the lambda.
    return fut.result()

# Create an executor that can run at most one function and run wait_on_future
with ThreadPoolExecutor(max_workers=1) as executor:
    fut = executor.submit(wait_on_future, executor)
    print (fut.result())
```

implicite, mais l'accès explicite (JavaScript, Python), les deux synchronisations peuvent exister [18].

4.3 Outils de synchronisation des *futures*

Nous avons précédemment présenté trois primitives permettant de manipuler des *futures* : `get`, `await` et `chain`. Nous allons à présent étudier ces trois constructions plus en détail.

`get` `get` est la construction la plus répandue pour manipuler les *futures*. `get` effectue une synchronisation bloquante : le *thread* appelant est bloqué jusqu'à ce que le *future* soit résolu, et retourne la valeur de résolution. On a donc une garantie forte, celle d'obtenir une valeur exploitable (sauf en cas de *deadlock*). De plus, le point de synchronisation est rendu extrêmement visible, et permet au programmeur de plus facilement détecter les éventuels *deadlocks* (par exemple si une fonction attend sur le *future* qu'elle doit résoudre). Le comportement de `get` rend cependant l'opération dangereuse dans des langages sans *multithreading* "conventionnel". JavaScript est *monothreadé* sans préemption, utiliser `get` n'aurait aucun sens, c'est pourquoi le langage propose uniquement le chaînage. Dans un langage à objets actifs, `get` empêche un objet de traiter d'autres messages, et il est facile de créer un cycle de *futures* pour produire un *deadlock*. Enfin, en Python, les *futures* sont obtenus par l'utilisation d'un *executor*, une entité possédant un *pool* de *threads* pour exécuter des fonctions asynchrones. Le code 15 illustre comment créer un *deadlock via* l'utilisation d'un *executor*.

`await` `await` est une alternative à `get` dans les langages à objets actifs. Comme nous avons pu le voir avec l'exemple du *broker*, `await` permet de facilement passer au traitement d'un autre message, au lieu d'attendre sur un *future*. En contrepartie, `await` ajoute un surcoût en temps et en mémoire. Son utilisation est étroitement liée au contexte. Dans le cas du *broker*, c'était l'aspect "traitement efficace de milliers de requêtes" qui faisait d'`await` une solution peu intéressante. Dans le cas d'un appel asynchrone

18. Il est possible d'avoir des *futures* explicitement typés et implicites d'accès, mais à notre connaissance aucun langage ne les implémente

isolé, très long, `await` peut être préférable à `get`, afin de laisser l'objet actif traiter d'autres messages. On peut également se poser la question de l'intérêt d'`await` dans des langages comme C++ ou Java, qui présentent un *multithreading* permissif, ainsi que de la préemption.

`chain` `chain` permet d'attacher un *callback* à la résolution d'un *future*. On peut ainsi conserver l'aspect asynchrone lié aux *futures*, en évitant les soucis liés à la synchronisation avec `get` ou `await` : au lieu de bloquer avec `get` jusqu'à obtenir la valeur pour effectuer des opérations sur celle-ci, on peut exécuter, toujours de façon asynchrone, une réaction ; et au lieu d'avoir un surcoût de changement de contexte dû à `await`, on reste dans un contexte déjà établi. En contrepartie, le flot d'exécution devient beaucoup plus complexe, il peut être nécessaire d'effectuer une opération de *bind* sur la fonction de *callback* pour capturer une partie de l'environnement courant. L'implémentation de `chain` dans des langages avec *futures* implicites d'accès (MultiLisp) paraît délicate, étant donné que ces *futures* sont transparents pour le programmeur, et sont confondus avec des objets non *futures*.

4.4 Comparaison des *futures* dans différents langages

Nous présentons ici un comparatif des *futures* dans différents langages, selon l'aspect implicite / explicite, la synchronisation *control-flow* / *dataflow* et les outils de synchronisation disponibles. La table 1 présente ce comparatif sur plusieurs langages que nous avons mentionnés ou étudiés.

Table 1 – Comparaison des *futures* dans plusieurs langages

Langage	Explicite / implicite	Synchronisation	Outils
C++	Explicites par création (`async`), typage et accès	Control-flow ; dataflow partielle possible *via* les *templates*	`get`
Java	Explicites par création (via un *executor*), typage et accès	Control-flow	`get`
ProActive (Java)	Implicites par création, accès et typage	Dataflow	Aucun
Python	Explicites par création (via un *executor*) et accès, implicites par typage	Control-flow, avec possibilité d'implémentation dataflow	`get`, `chain`
Multilisp	Explicites par création (`future`), implicites par accès et typage	Dataflow	Aucun
JavaScript	Explicites par création et accès, implicites par typage	Dataflow	`chain`
Encore	Explicites par création, accès et typage	Control-flow	`get`, `await`, `chain`, `forward`, combinateurs
ABS	Explicites par création (opérateur `!`), accès et typage	Control-flow	`get`, `await`
DeF	Implicites par création, explicites par accès et typage	Dataflow	`get`

Nous pouvons constater qu'une grande quantité de langages utilisent des *futures* explicites, au moins au niveau du typage. De ce fait une très grande quantité de langages offrent une synchronisation *control-flow*. `get` est une construction qui est presque systématiquement présente, `chain` est moins représentée, et `await` est plus réservée aux langages à objets actifs. On note également qu'un seul langage propose des *futures* explicites avec une synchronisation *dataflow*, à savoir DeF. De même, un seul langage propose la construction `forward`, à savoir Encore. Bien que de nouvelles constructions apparaissent, les *futures* explicites *control-flow* sont majoritaires.

5 Conclusion

Les *futures* sont une construction intéressante en programmation concurrente, qui permettent l'écriture d'un code visuellement plus séquentiel, et résolvent des problèmes de synchronisation, comme la transmission de résultats entre tâches parallèles. Leur utilité ne se limite pas qu'à cela, comme nous l'avons vu, les *futures* peuvent être utilisés pour modéliser le concept de réaction à un évènement, par exemple en JavaScript.

La programmation par objets actifs, idéale dans le contexte de systèmes distribués, tire profit des *futures* pour implémenter son mode de communication par appels de fonctions asynchrones. Diverses constructions basées sur les *futures*, et tirant partie de leur aspect asynchrone, ont été créées, comme *ParT*. D'autres construction, comme `forward`, résolvent des problématiques liées aux *futures*, comme la délégation.

Certains axes restent à explorer, notamment l'implémentation des *dataflow explicit futures* d'Henrio, qui permettraient de combiner les avantages de deux mondes : l'aspect contrôle de la synchronisation inhérent aux *futures* explicites, et la légèreté syntaxique, inhérente à la synchronisation *dataflow*. Une tentative d'implémentation de *DeF* dans le langage Encore est actuellement en cours.

Références

[1] Gul Agha. *Actors : A model of concurrent computation in distributed systems.* MIT Press, 1986.

[2] Saba Alimadadi, Di Zhong, Magnus Madsen, and Frank Tip. Finding broken promises in asynchronous javascript programs. *Proc. ACM Program. Lang.*, 2(OOPSLA) :162 :1–162 :26, October 2018.

[3] Laurent Baduel, Françoise Baude, Denis Caromel, Arnaud Contes, Fabrice Huet, Matthieu Morel, and Romain Quilici. *Programming, Composing, Deploying for the Grid*, pages 205–229. Springer London, London, 2006.

[4] Henry C. Baker, Jr. and Carl Hewitt. The incremental garbage collection of processes. In *Proceedings of the 1977 Symposium on Artificial Intelligence and Programming Languages*, pages 55–59, New York, NY, USA, 1977. ACM.

[5] Frank De Boer, Vlad Serbanescu, Reiner Hähnle, Ludovic Henrio, Justine Rochas, Crystal Chang Din, Einar Broch Johnsen, Marjan Sirjani, Ehsan Khamespanah, Kiko Fernandez-Reyes, and Albert Mingkun Yang. A survey of active object languages. *ACM Comput. Surv.*, 50(5) :76 :1–76 :39, October 2017.

[6] Stephan Brandauer, Elias Castegren, Dave Clarke, Kiko Fernandez-Reyes, Einar Broch Johnsen, Ka I. Pun, S. Lizeth Tapia Tarifa, Tobias Wrigstad, and Albert Mingkun Yang. *Parallel Objects for Multicores : A Glimpse at the Parallel Language Encore*, pages 1–56. Springer International Publishing, Cham, 2015.

[7] Denis Caromel, Ludovic Henrio, and Bernard Paul Serpette. Asynchronous and deterministic objects. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, pages 123–134, New York, NY, USA, 2004. ACM.

[8] Sylvan Clebsch and Sophia Drossopoulou. Fully concurrent garbage collection of actors on many-core machines. *SIGPLAN Not.*, 48(10) :553–570, October 2013.

[9] Kiko Fernandez-Reyes, Dave Clarke, Elias Castegren, and Huu-Phuc Vo. Forward to a promising future. In Giovanna Di Marzo Serugendo and Michele Loreti, editors, *Coordination Models and Languages*, pages 162–180, Cham, 2018. Springer International Publishing.

[10] Kiko Fernandez-Reyes, Dave Clarke, and Daniel S. McCain. Part : An asynchronous parallel abstraction for speculative pipeline computations. In Alberto Lluch Lafuente and José Proença, editors, *Coordination Models and Languages*, pages 101–120, Cham, 2016. Springer International Publishing.

[11] Robert H. Halstead, Jr. Multilisp : A language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7(4) :501–538, October 1985.

[12] Ludovic Henrio. Data-flow Explicit Futures. Research report, I3S, Université Côte d'Azur, April 2018.

[13] Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte, and Martin Steffen. Abs : A core language for abstract behavioral specification. In Bernhard K. Aichernig, Frank S. de Boer, and Marcello M. Bonsangue, editors, *Formal Methods for Components and Objects*, pages 142–164, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[14] David Kitchin, Adrian Quark, William R. Cook, and Jayadev Misra. The orc programming language. In *FMOODS/FORTE*, 2009.

[15] Magnus Madsen, Ondřej Lhoták, and Frank Tip. A model for reasoning about javascript promises. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA) :86, 2017.

[16] Daniel Sean McCain. Parallel combinators for the encore programming language. Master's thesis, Uppsala University, Department of Information Technology, 2016.

[17] Derek Wyatt. *Akka Concurrency*. Artima, 2013.