Une implémentation des *dataflow explicit futures*

Amaury MAILLÉ

28 août 2019

Table des matières

1 Introduction **1**
1.1 *Futures* implicites et explicites . . . 1
1.2 *Futures control-flow* et *dataflow* . . . 1
1.3 *Dataflow explicit futures* . . . 2
1.3.1 Godot . . . 3
1.3.2 Encore . . . 3
1.4 Contributions . . . 3

2 Mise en contexte **4**
2.1 Une comparaison des *futures* implicites, explicites, *control-flow* et *dataflow* . . . 4
2.2 Acteurs et objets actifs . . . 5
2.3 Le langage Encore . . . 6
2.3.1 Les *futures* d'Encore . . . 7
2.3.2 Processus de compilation d'Encore . . . 7
2.3.3 Généricité dans Encore . . . 11
2.3.4 Une introduction au *runtime* d'Encore . . . 12
2.3.5 Une illustration du fonctionnement du *runtime* d'Encore . . . 13
2.4 Une introduction informelle à Godot . . . 14
2.4.1 Sous-typage et *collapse* . . . 14
2.4.2 Sémantique de `get*` . . . 15
2.4.3 Remarques . . . 16

3 Une implémentation des *dataflow explicite futures* sans introspection native **16**
3.1 Implémentation des *dataflow explicit futures* dans Encore . . . 16
3.1.1 Syntaxe . . . 17
3.1.2 Typage . . . 18
3.1.3 *Runtime* et génération de code . . . 19
3.1.4 Introspection & Synchronisation . . . 21
3.2 Le cas d'`async*` et des fonctions globales . . . 24
3.3 Le cas des génériques . . . 25
3.3.1 Les génériques dans le compilateur . . . 25
3.3.2 Solutions envisagées . . . 26
3.3.3 Fonctions et classes génériques . . . 28
3.4 *Compile-time expansion* des génériques . . . 28
3.4.1 Solution retenue . . . 29
3.4.2 Implémentation . . . 30
3.4.3 Généralisation . . . 33

4 Travaux liés **34**

5 Conclusion **35**

A Les *futures* comme *framework* d'échange de messages **37**

B *Futures* implicites et explicites **37**

C Traduction en C d'un appel asynchrone Encore **38**

D Factorielle en Encore avec `async*` **40**

E Fonction Haskell `checkOneBindingForFlow` **41**

Table des figures

1 AST Encore issu du code en listing 6 10
2 AST C après traitement de l'AST Encore en figure 1 10
3 Graphe des appels C déclenchés par un appel asynchrone Encore 14
4 AST Encore correspondant au listing 14 avant *lift* 19
5 AST Encore correspondant au listing 14 après *lift* 19

Liste des tableaux

1 Comparaison du lancement de tâche asynchrone et réception de valeur entre MPI et les *futures* 37
2 Illustration de la différence explicite / implicite sur les aspects création, accès et typage des *futures* 38

Listings

1 Implémentation d'un *broker* dans le langage à objets actifs Encore 5
2 Implémentation d'un *broker* parallèle dans le langage à objets actifs Encore . . . 6
3 Illustration de l'utilisation des *futures* dans Encore 8
4 Opérateur `async` dans Encore avant *desugaring* 8
5 Opérateur `async` dans Encore après *desugaring* 8
6 Invocation asynchrone de méthode dans Encore 9
7 Traduction C du code Encore en listing 6 10
8 Illustration du mécanisme de *compile-time expansion* des génériques (C++) . . . 11
9 Illustration du mécanisme de *type erasure* des génériques (Encore) 11
10 Code Encore d'exemple pour le fonctionnement de l'envoi de message, *i.e* l'appel asynchrone 12
11 Fonctionnement de `get*` dans Godot 15
12 Factorielle asynchrone avec `Flow` dans Encore 17
13 Implémentation de la fonction `collapseFlow` en Haskell 19
14 Illustration d'un cas de *lift* en Encore 19
15 Implémentation de `get*` en C 24
16 Implémentation de `spawn_star` en Encore 24
17 Implémentation de `spawn` en Encore 24

18 Implémentation du `TaskRunner` en Encore 24
19 Utilisation de `Flow` avec les génériques 29
20 Algorithme permettant de déterminer si un paramètre générique devient `Flow` . . 33
21 Traduction C d'un appel asynchrone Encore 39
22 Envoi C d'un message 39
23 Traitement C d'un appel asynchrone 40
24 Factorielle asynchrone en Encore avec `async*` 40
25 Fonction `checkOneBindingForFlow` 42
26 Exemple de *bind* de type générique pour un paramètre de type statique `t` 42
27 Exemple de *bind* de type générique pour un paramètre de type *builtin* 42
28 Exemple de *bind* de type générique pour un paramètre de type classe paramétrée 43
29 Exemple de *bind* de type générique pour un paramètre de type continuation . . . 43

1 Introduction

En programmation asynchrone, un *future* [4] est un objet, initialement vide, destiné à recevoir le résultat d'un calcul. Un *future* est associé à une tâche asynchrone : le lancement d'une tâche produit un *future* vide. Lorsque la tâche produit une valeur `v`, cette valeur est placée dans le *future* qui lui est associé. On dit alors que le *future* est résolu avec la valeur `v`. Lorsqu'un *future* est destiné à être résolu par une valeur de type `T`, on parle de « *future* sur `T` ». Nous désignons le type « *future* sur le type `T` » par la notation `Fut[T]`.

Les *futures* sont dotés d'un mécanisme de synchronisation, c'est-à-dire qu'une tentative d'accès à la valeur contenue dans un *future* bloquera jusqu'à ce qu'il soit résolu. La synchronisation est en général déclenchée au travers d'une opération, traditionnellement nommée `get`, qui reçoit en paramètre le *future* à synchroniser, et renvoie la valeur de résolution de ce *future*.

Nous commençons par présenter les deux classifications existantes de *futures* : *futures* implicites ou explicites, et *futures control-flow* ou *dataflow*.

1.1 *Futures* implicites et explicites

La distinction entre *futures* explicites et implicites se base traditionnellement sur leur typage[5]. Les *futures* d'un langage sont considérés comme explicites lorsqu'il existe un type dédié pour les représenter. Par exemple, en C++, qui possède un typage statique, on trouve la classe `future<T>` ; en Python, qui possède un typage dynamique, on trouve la classe `Future`. Ainsi, il n'est pas possible de manipuler un *future* sur `T` de la même manière qu'une instance de `T`. Par exemple, l'opérateur + ne fonctionnera pas sur un *future* sur `int`. [1]

A l'inverse, les *futures* implicites sont caractérisés par une absence de type dédié. Dans un langage à *futures* implicites, une variable de type `T` peut présenter un *comportement de future*. Par exemple, dans le langage MultiLisp, l'expression `(+ f 1)`, qui renvoie la somme de la valeur de `f` et 1, fonctionne que `f` soit un `int` ou un *future* sur `int`. Si `f` n'est pas un *future*, le calcul est effectué immédiatement. Si `f` est un *future*, le calcul est mis en attente jusqu'à ce que `f` soit résolu.

Dans la suite de ce rapport, nous nous restreindrons à un sous-ensemble des langages à *futures* explicites : les langages à typage statique dans lesquels les *futures* sont définis comme types paramétriques, par exemple C++ et sa construction `future<T>`.

L'aspect explicite (resp. implicite) de la synchronisation est lié à l'aspect explicite (resp. implicite) des *futures*. En C++ ou en Python, les *futures* présentent une opération de synchronisation, `get` en C++, `result` en Python. Cette opération doit être effectuée explicitement par le programmeur. A l'inverse, en MultiLisp, l'expression `(+ f 1)` peut provoquer une synchronisation si `f` est un *future*. Cette synchronisation sera déclenchée automatiquement et est donc implicite.

1.2 *Futures control-flow* et *dataflow*

Henrio propose une seconde classification des *futures*, basée sur la synchronisation [11]. Considérons un *future* `f` dont le résultat est un *future* sur `int`. Nous verrons en section 2.2, dédiée aux objets actifs, dans quelles circonstances la construction de *futures* imbriqués peut apparaître.

Une distinction importante est établie entre les concepts de « valeur » et les « *future* ». On peut le voir ainsi : un *future* n'est pas considéré comme une valeur. Considérons à présent que nous effectuons une synchronisation sur `f`, au moyen de `get`. Deux comportements peuvent être envisagés :

1. La distinction explicite / implicite peut s'effectuer de façon plus fine. L'annexe B entre plus dans le détail

1. `get` attend, si besoin, que `f` soit résolu, en l'occurrence avec un *future* (que nous nommons `f2`). Une fois `f` résolu, `get` renvoie `f2`. Il est nécessaire d'effectuer un second `get` pour obtenir un `int` à partir de `f2`. Henrio nomme ce comportement, où chaque synchronisation résout un « niveau » de *future*, « synchronisation *control-flow* » ;
2. `get` attend, si besoin, que `f` soit résolu avec une valeur. Comme `f` est un *future* sur *future* sur `int`, il ne peut pas être résolu par une valeur ; il peut uniquement être résolu par un *future*. Nommons ce *future* `f2`. `get` va alors attendre que `f2` soit résolu avec une valeur `v`, ici de type `int`. Une fois les deux *futures* résolus, `get` renvoie la valeur `v`. Henrio nomme ce comportement, où une synchronisation déclenche d'autres synchronisations jusqu'à obtenir une valeur, « synchronisation *dataflow* ».

Henrio observe une corrélation entre l'aspect implicite et explicite des *futures* et le type de synchronisation utilisé dans les langages existants. En particulier, la synchronisation *dataflow* est propre aux *futures* implicites et la synchronisation *control-flow* est propre aux *futures* explicites. Plus précisément, ce sont les propriétés mêmes du caractère implicite (resp. explicite) qui permettent la mise en place de la synchronisation *dataflow* (resp. *control-flow*). En effet :

- La synchronisation *dataflow* doit pouvoir attendre sur une chaîne de *futures* de longueur indéterminée (*e.g*, appels récursifs de tâches asynchrones). Une telle chaîne ne peut être typée à l'aide de *futures* paramétrés (explicites) ;
- L'écriture d'une tâche qui renvoie une valeur de type `T` ou qui délègue à une autre tâche asynchrone la responsabilité de produire une valeur n'est pas possible avec des *futures* explicites. Si la tâche délègue à une autre tâche asynchrone, alors elle doit être typée `Fut[T]`. Si la tâche souhaite produire une valeur de type `T`, elle doit être typée `T`. S'il n'existe pas de conversion de `T` vers `Fut[T]`, une telle fonction ne peut pas être écrite avec des *futures* explicites. Dans le cas des *futures* implicites, ce problème ne se présente pas ;
- La synchronisation *control-flow* doit effectuer exactement une seule synchronisation. Dans un langage typé statiquement il est donc nécessaire que les *futures* possèdent un type dédié. Comparez `Fut[Fut[Foo]]` et `Foo`. Le premier cas expose clairement deux *futures* : un premier dont le résultat est une valeur de type « *future* sur `Foo` », et un second dont le résultat est une valeur de type `Foo`. Il y a donc deux synchronisations successives nécessaires pour accéder à la valeur. Dans le seconde cas, seul le type `Foo` apparaît. Si l'on souhaitait effectuer une unique synchronisation, quel en serait le type résultant ? Comment déterminer si une variable de type `Foo` présente un comportement de *future* ? Est-ce un *future* à un seul niveau ? Deux ? *N* ? La difficulté de répondre à ces questions explique la prévalence de la synchronisation *dataflow* sur les *futures* implicites.

Toujours dans [11], Henrio montre que cette corrélation n'est pas absolue, et propose la construction *DeF* (*Dataflow explicit Futures*).

1.3 *Dataflow explicit futures*

Nous adoptons les notations du système de types Godot, que nous allons présenter dans cette section. `Fut[T]` dénote un *future* explicite ***control-flow*** sur un type `T`. `Flow[T]` dénote un *future* explicite *dataflow* sur un type `T`. `get*` est la fonction `get` spécialisée sur les `Flow[T]`.

Les *dataflow explicit futures* sont des *futures* explicites présentant une synchronisation *dataflow*. Nous adaptons la notation d'Henrio et désignons un *dataflow explicite future* sur le type `T` par la notation `Flow[T]` de Godot. L'union des *futures* explicites et de la synchronisation *dataflow* implique qu'un `Flow[T]` présente certaines propriétés des *futures* implicites, afin de permettre la synchronisation *dataflow* :

- Tout type `T` est sous-type de `Flow[T]`. Par exemple, il est possible d'écrire `Flow[int] f = 1` cela permet de typer une tâche dont le résultat est soit un *future* sur `T`, soit un `T` ;

- Tout type `Flow[Flow[T]]` est simplifié en le type `Flow[T]`. Cette simplification s'effectue de façon récursive et, couplée à la transformation précédente, permet de typer une tâche récursive terminale asynchrone.

Toujours dans [11], Henrio propose un système de types formalisant ces deux règles, ainsi que la version *dataflow explicit* de la méthode `get`. Ces travaux ont été repris et complétés par Fernandez-Reyes et al. dans [9], qui, entre autres, étendent le système de types d'Henrio aux types paramétrés, et proposent une sémantique des autres opérations usuelles sur les *futures* explicites. Ce nouveau système de types s'intitule Godot.

1.3.1 Godot

Godot propose de formaliser le fonctionnement des *dataflow explicit futures* au travers de deux règles de typage. Là où une valeur de type `Flow[T]` est attendue, il est possible de passer une valeur de type `T`, ou une valeur de type `Flow[Flow[T]]`. Par exemple `Flow[int] f = 3` est correctement typé selon Godot. Ces deux règles couvrent les deux propriétés des *futures* implicites que les *dataflow explicit futures* doivent présenter, identifiées par Henrio.

Godot formalise également le fonctionnement de `get*`. Godot suppose l'existence une fonction `isFlow` permettant de savoir, *à l'exécution*, pour une valeur `v` s'il s'agit d'un `Flow` ou non. Godot suppose donc l'existence de capacités d'introspection. Le fonctionnement de `get*` est alors trivial : `get*` se comporte initialement comme `get`, et attend que le `Flow` soit résolu ; si la valeur de résolution n'est pas un `Flow`, `get*` la renvoie ; sinon, `get*` s'appelle récursivement sur la valeur de résolution.

1.3.2 Encore

Comme nous l'établirons plus formellement en section 2.2, les *futures* apparaissent naturellement dans les langages à objets actifs. Dans ces langages, des entités *monothreadées* s'exécutant en parallèle communiquent par appels asynchrones de fonctions. Le langage Encore est un de ces langages. Fernandez-Reyes et al. ont grandement contribué au développement du compilateur d'Encore, *encorec* [1], ont publié une longue étude du langage ([6]), et y ont implémenté plusieurs des constructions qu'ils ont pu développer en rapport avec les *futures* [10, 8]. Encore possède des *futures* explicites *control-flow* et son compilateur est assez simple pour envisager une implémentation des *dataflow explicit futures*.

1.4 Contributions

Ce rapport présente nos principales contributions :

- Une implémentation de Godot dans le langage Encore, dépourvu des capacités d'introspection supposées par Godot ;
- Une identification des problématiques soulevées par cette absence d'introspection ;
- Nos approches pour répondre aux problématiques liées à l'absence d'introspection.

Nous présentons en section 2 des concepts centraux afin que le lecteur puisse se familiariser avec ceux-ci : une comparaison des *futures* explicites et implicite, et une comparaison des synchronisation *control-flow* et *dataflow*, afin que le lecteur puisse mesurer l'intérêt des *dataflow explicit futures* ; les langages à objets actifs, dont le langage Encore est un représentant ; le langage Encore, ses *futures*, son processus de compilation, son implémentation des concepts de généricité et son environnement d'exécution (tous ces éléments seront intensivement mentionnés dans notre contribution) ; enfin, une introduction plus formelle à Godot. La section 3 présente nos contributions. La section 4 présente des travaux similaires aux nôtres. Enfin, la section 5 présente nos conclusions.

2 Mise en contexte

Nous présentons ici les concepts critiques à la compréhension du présent rapport, en particulier l'intérêt de la construction *dataflow explicit futures* et des concepts mentionnés dans notre contribution. En premier lieu, nous présentons une comparaison des forces et faiblesses des *futures* explicites et implicites, ainsi qu'une comparaison des avantages et inconvénients des synchronisations *control-flow* et *dataflow*. Nous présentons ensuite la programmation par objets actifs et ses concepts fondamentaux. Une troisième section présente le langage Encore, ses *futures*, son processus de compilation, son implémentation de la généricité et son environnement d'exécution. Une quatrième section présente formellement le système de types et de calcul de Godot.

2.1 Une comparaison des *futures* implicites, explicites, *control-flow* et *dataflow*

Les *futures* explicites bénéficient d'un type dédié, tel que `future<T>` en C++, ou `Future` en Python. Ils bénéficient également d'opérations dédiées, telles que `get` ou `then`. Les *futures* implicites ne bénéficient pas d'un type ou d'opérations dédiées, et sont confondus avec des variables de type `T`. Ces variables présentent un comportement de *future*.

Les *futures* implicites permettent d'écrire du code plus réutilisable, car une variable de type `T` peut indifféremment recevoir une valeur de type `T` ou une valeur de « type » *future* sur `T`. L'absence de primitive de synchronisation renforce la réutilisation de code. En contrepartie, l'aspect implicite de la synchronisation rend complexe la détection et la résolution de *deadlocks*, car le programmeur n'est pas nécessairement à même de savoir quelles valeurs sont *futures* ou non. Par conséquent, une opération sur une valeur de type `T` peut être bloquante ou non, et le programmeur ne sera pas non plus à même de le déterminer à l'avance.

A l'inverse, les *futures* explicites limitent la réutilisation de code car il n'existe (généralement) pas de conversion triviale d'un type `T` vers un type *future* sur `T`. La réutilisation est d'autant plus amoindrie par la nécessité d'une synchronisation explicite. En contrepartie, cette synchronisation explicite et cette distinction des types offrent au programmeur un contrôle complet sur son programme. Si une fonction prend en paramètre une valeur de type `T`, il n'y aura pas d'ambiguïté sur est-ce que cette valeur se comporte comme un *future* ou non. Les synchronisations seront rendues explicites au moyen de la fonction `get`. Par ailleurs, la présence d'un type *future* permet de doter les *futures* d'outils supplémentaires : chaînage, création de *pipelines* parallèles [10], délégation de résolution [8]...

La question de la synchronisation se pose sur les *futures* imbriqués. La synchronisation *control-flow* attend que le niveau le plus extérieur de *future* soit résolu, et la synchronisation *dataflow* attend que tous les niveaux soient résolus, récursivement.

La synchronisation *control-flow* est utile lorsqu'il est nécessaire de déterminer si une tâche est achevée. Par exemple, dans un ordonnanceur, il est plus pertinent de savoir que la tâche A est terminée plutôt que savoir que la chaîne de tâches lancée par A est terminée. L'ordonnanceur souhaite lancer des tâches. Ce qu'elles font une fois lancées ne le concerne en rien.

La synchronisation *dataflow* est utile de façon plus générale. Pour un développeur, savoir qu'un unique `get` fournira une valeur exploitable, et non un *future* est très pratique. Lorsqu'un développeur lance un calcul asynchrone, il est plus intéressé par le résultat du calcul, plutôt que par le nombre de synchronisations nécessaire pour obtenir ce résultat. Le code à écrire peut être substantiellement réduit.

Listing 1 – Implémentation d'un *broker* dans le langage à objets actifs Encore

```
1  active class Broker {
2      var workers : Worker[]
3      var worker : int
4
5      def submit(job : Job[t]) : t {
6          var worker = workers[++worker % workers.size()]
7          var res : Fut[t] = worker!process(job)
8          return get(res)
9      }
10 }
```

Unifier la synchronisation *dataflow* et les *futures* explicites offrirait les avantages de deux mondes : la maîtrise du déclenchement des synchronisations et la présence d'opérations dédiées sur les *futures*, induites par l'aspect explicite ; la garantie d'obtenir une valeur exploitable en un unique `get`, la réutilisation de code et la possibilité de typer des chaînes non statiquement bornées de *futures*, induites par la synchronisation *dataflow*

2.2 Acteurs et objets actifs

La programmation par acteurs [3] est un paradigme de programmation dans lequel le programmeur manipule des entités *monothreadés*, qui s'exécutent en parallèle les unes des autres communiquent entre elles par passages de messages. La programmation par objets actifs [5] est un cas particulier de la programmation acteur, où les entités sont des objets (au sens « programmation orientée objet ») et où les passages de messages correspondent à des appels de méthodes.

Les appels de méthodes sur des objets actifs sont asynchrones. Lorsqu'un objet actif appelle une méthode `m` d'un objet actif `o`, le résultat de cet appel est un *future* sur le résultat de l'exécution de `m`. Chaque objet actif maintient une file de messages, et les exécute l'un après l'autre. C'est dans ce contexte que des *futures* imbriqués peuvent apparaître.

Un exemple Considérons le principe d'un *broker*. Un *broker* est une entité qui reçoit des travaux (*Jobs*) et doit les répartir entre plusieurs objets qui vont exécuter ces travaux (*Workers*). Le *broker* dispose d'une méthode `submit` et d'un ensemble (*pool*) de *workers*. La méthode `submit` reçoit en paramètre un travail, et le transmet à un *worker* selon une politique définie. Les *workers* possèdent une méthode `process`, qui prend en paramètre le travail à exécuter, et renvoie le résultat de l'exécution de ce travail. Considérons une implémentation du *broker* dans le langage Encore en listing 1, basée sur l'exemple proposé par Fernandez-Reyes et al. dans [8].

Analyse L'implémentation de `submit` est simple : récupérer un *worker* selon une politique *round-robin*, lui faire exécuter le travail, et renvoyer le résultat de ce travail une fois qu'il est disponible. L'opérateur `!` dénote l'invocation asynchrone de méthode [2]. Cette implémentation, naturelle en programmation orientée objets, est inefficace en programmation par objets actifs. Comme le *broker* est *monothreadé*, chaque appel à `submit` sur une même instance de la classe `Broker` sera mis en suspens jusqu'à ce que les précédents appels soient terminés. Cette implémentation ne permet donc pas d'exécuter plusieurs travaux en parallèle.

2. Il s'agit d'une contrainte du langage. Le compilateur d'Encore lèvera une erreur si l'opérateur d'invocation synchrone (`.`) est utilisé sur un objet actif

Listing 2 – Implémentation d'un *broker* parallèle dans le langage à objets actifs Encore

```
1  active class Broker {
2      var workers : Worker[]
3      var worker : int
4
5      def submit(job : Job[t]) : Fut[t] {
6          var worker = workers[++worker % workers.size()]
7          var res : Fut[t] = worker!process(job)
8          return res
9      }
10 }
```

Une alternative Considérons une solution alternative dans le listing 2. Au lieu d'effectuer `get` sur le *future* obtenu par appel à `process`, `submit` renvoie ce *future* et laisse l'utilisateur effectuer le `get` lui-même. Le type de retour de `submit` devient donc `Fut[t]`. Or, les appels à `submit` sont eux-mêmes asynchrones, le type de retour d'un appel asynchrone à `submit` est donc `Fut[Fut[t]]`. D'une façon générale, les *futures* imbriqués apparaissent lorsqu'il est nécessaire de conserver l'aspect asynchrone tout en enchaînant des appels entre objets actifs. Ce fort couplage du type de retour des fonctions avec la structure logique du code est délicat à maintenir et à manipuler, et l'utilisation des `Flow` serait une solution intéressante.

2.3 Le langage Encore

Le langage Encore [6] est un langage multi-paradigmes, permettant de la programmation fonctionnelle et de la programmation par objets actifs. Il s'agit du langage dans lequel nous avons effectué une implémentation de `Flow`, aussi nous le présentons ici afin de familiariser le lecteur avec Encore. Nous commençons par une présentation générale du langage, puis nous continuons sur une présentation de ses *futures* et des outils pour les manipuler ; une présentation de son mécanisme de compilation sur lequel nous nous appuierons en section 3 ; et une présentation de son implémentation des génériques sur laquelle nous nous appuierons en section 3.3.

Le langage Encore est un langage à typage statique fort (les type des variables sont apparents dans le code, et il n'y a pas de conversion implicite des objets [3]), qui propose des constructions de langages fonctionnels (*pattern matching*, *datatypes*, continuations), de langages orientés objets (héritage, interfaces) ainsi que de la généricité sur les classes et les fonctions. Le langage Encore reste avant tout un langage à objets actifs et propose nativement une implémentation des *futures*, ainsi que les primitives pour les manipuler : `get`, `await` et `then` [4]. Le langage Encore propose également la construction `async`, qui permet de lancer une fonction de façon asynchrone et produit un *future* sur le résultat de cette fonction. Enfin, l'opérateur `!` que nous avons vu dans l'exemple du *broker* permet d'invoquer une méthode de façon asynchrone sur un objet actif.

L'unique implémentation existante d'Encore compile vers du code machine en deux temps : une première compilation vers du code C, puis une compilation du code C vers du code machine. Le compilateur d'Encore, *encorec*, écrit en Haskell, s'occupe de la traduction du code Encore vers le code C. Le compilateur *Clang* se charge de la compilation du code C vers le code machine. Le passage vers du code C permet une simplification du processus de compilation, en déléguant la compilation du C vers l'assembleur à un compilateur préexistant. Cela permet

3. Par exemple `"5" + 2` n'est pas une expression Encore valide, mais est valide en JavaScript qui possède un typage plus faible

4. Bien que Godot [9] présente une sémantique d'`await` et de `then` utilisant `Flow`, nous ne reviendrons pas sur ces dernières

aux développeurs du compilateur d'Encore de ne pas avoir à réimplémenter les (nombreuses) optimisations effectuées par Clang. En particulier, les *futures* et les opérations sur les *futures* sont codées directement en C, ce qui offre des garanties de performances. La gestion des aspects objets actifs est réalisée à l'aide du *runtime* du langage Pony [7], un langage de programmation multi-paradigme, permettant de la programmation par acteurs, et dont le *runtime* est écrit en C.

2.3.1 Les *futures* d'Encore

Nous présentons brièvement le fonctionnement des *futures* et les outils à la disposition du programmeur pour les créer et les manipuler. Cette présentation a pour objectif de familiariser le lecteur avec les outils et notations disponibles, de façon à ce qu'il puisse effectuer une comparaison avec nos choix d'implémentation de Godot.

Les *futures* d'Encore sont explicites *control-flow*, dénotés par le type `Fut[T]` qui représente un *future* sur `T`. L'aspect explicite *control-flow* indique que la construction `Fut[Fut[int]]` représente un *future* sur *future* sur `int`.

Il est possible de créer des *futures* de deux façons : soit au travers de l'opérateur `async`, soit au travers de l'opérateur `!`.

Opérateur `async` L'opérateur `async` permet de lancer une fonction de façon asynchrone. Si le type de la fonction est `T`, le *future* obtenu via `async` est typé `Fut[T]`.

Opérateur `!` L'opérateur `!` permet de lancer une méthode d'un objet actif de façon asynchrone. Si le type de la méthode est `T`, le *future* obtenu via l'opérateur `!` est typé `Fut[T]`.

La fonction `get` prend en paramètre un *future* sur `T` et renvoie une valeur de type `T`. L'aspect explicite *control-flow* des *futures* d'Encore interdit à `get` d'effectuer une récursion. Considérons une variable `f` de type `Fut[Fut[int]]`. `get(f)` renverra une valeur de type `Fut[int]` et il sera nécessaire d'effectuer un second `get` pour obtenir un `int`.

Le listing 3 illustre ces éléments. La ligne 20 montre une invocation asynchrone de fonction globale au travers d'`async`. Les lignes 21 et 22 montrent deux invocations asynchrones de méthodes sur un objet actif. On peut observer à la ligne 22 qu'il est nécessaire d'effectuer deux `get` pour obtenir une valeur exploitable depuis le *future* renvoyé par l'expression `x!g()`.

2.3.2 Processus de compilation d'Encore

Le processus de compilation fait intervenir trois concepts : le *front-end*, la représentation intermédiaire (*intermediate representation* ou IR) et le *back-end*. La représentation intermédiaire est la façon que le compilateur a de représenter le code en interne. Le *front-end* réalise le travail de construction de la représentation intermédiaire. Le *back-end* réalise le travail de traduction de la représentation intermédiaire en langage cible.

front-end Le *front-end* fait intervenir trois phases : le *parsing*, le *desugaring* et le *typechecking*.

Parsing & lexing Le *parsing* (couplé avec le *lexing*) consiste en la lecture du code source, afin de détecter les erreurs de syntaxe et de construire un Arbre de Syntaxe Abstrait (*Abstract Syntax Tree* ou AST). L'AST représente le code sous forme d'arbre, qui est une structure pratique à manipuler (l'AST n'est pas nécessairement l'IR).

Listing 3 – Illustration de l'utilisation des *futures* dans Encore

```
1  import Task
2
3  fun f() : int
4      return 42
5  end
6
7  active class Foo
8      def f() : int
9          return 42
10     end
11
12     def g() : Fut[int]
13         return this!f()
14     end
15 end
16
17 active class Main
18     def main() : unit
19         var x = new Foo()
20         println(get(async(f()))) -- 42
21         println(get(x!f())) -- 42
22         println(get(get(x!g()))) -- 42
23     end
24 end
```

Desugaring Le *desugaring* est un ensemble de transformations appliqué à l'AST afin de remplacer le sucre syntaxique par son véritable code[5]. Par exemple, dans Encore, l'opérateur `async`, qui permet de lancer une fonction de façon asynchrone, est du sucre syntaxique autour d'un appel à la fonction `spawn`, qui prend en paramètre une *closure*. Nous illustrons cela dans les listings 4 et 5. Le premier montre le code initial (avec `async`), le second montre le « code » après *desugaring* (avec `spawn_star`)[6].

Listing 4 – Opérateur `async` dans Encore avant *desugaring*

```
1  import Task
2
3  fun id_int(x : int) : int
4      return x
5  end
6
7  fun g(x : int) : Fut[int]
8      return async(f(x))
9  end
```

Listing 5 – Opérateur `async` dans Encore après *desugaring*

```
1  import Task
2
3  fun id_int(x : int) : int
4      return x
5  end
6
7  fun g(x : int) : Fut[int]
8      var cls = fun() => f(x) end
9      return spawn_star(cls)
10 end
```

Typechecking Le *typechecking* consiste à s'assurer que le programme *type* correctement. Par exemple, en Encore, il n'est pas possible d'ajouter un entier à une chaîne de caractères.

5. Le sucre syntaxique est une construction permettant au programmeur d'écrire de façon plus simple un code complexe

6. Le *desugaring* ne modifie pas le code, il modifie la vision que le compilateur a du code

Listing 6 – Invocation asynchrone de méthode dans Encore

```
1 var x = 12
2 o!m(x)
```

L'expression `"hello" + 5` ne passerait donc pas le *typechecking*. Dans le cas d'Encore, cette vérification est faîte en annotant l'AST afin d'ajouter un type à chaque nœud dans un premier temps, puis en s'assurant que les types correspondent à ce qui est attendu. Le *typechecking* permet ainsi d'éviter un certain nombre d'erreurs.

back-end Le *back-end* génère le code dans le langage cible. Dans le cas d'Encore, ce processus se fait en deux étapes. L'AST initial (nommons le « AST Encore ») est parcouru pour créer un second AST (nommons le « AST C »). Chaque nœud de l'AST Encore est traduit en un ou plusieurs nœuds dans l'AST C, qui représentent la structure du code C à obtenir. Par exemple, considérons le nœud `MessageSend` dans l'AST Encore. Ce nœud correspond à un appel asynchrone de méthode. Les enfants de ce nœud sont *target*, *name* et *args*. *target* est (la représentation de) l'objet actif sur lequel la méthode est appelé. *name* est (la représentation de) la méthode appelée. *args* est une liste (des représentations) des arguments passés à la méthode. La traduction d'un appel de méthode asynchrone en C implique d'appeler plusieurs fonctions C, c'est-à-dire une séquence d'instructions. En conséquence, le nœud `MessageSend` de l'AST Encore est traduit dans l'AST C par le nœud `Seq` qui a pour enfant une liste (des représentations) des fonctions à appeler. À la traduction du nœud `Seq` le compilateur d'Encore sait qu'il doit générer un ensemble d'instructions, terminées par des points-virgule.

Nous illustrons ce processus. Considérons le code présent dans le listing 6. Ce code déclare une variable nommée `x`, de type `int` et lui affecte la valeur `12`. Ce code appelle ensuite, de façon asynchrone, la méthode `m` sur l'objet actif `o`, et lui passe la variable `x` en paramètre.

Le *parsing* de ce code produit l'AST Encore en figure 1. Le nœud `Seq` représente une succession d'expressions. Les labels `elems` indiquent la présence d'une liste. `MessageSend` correspond à un appel de méthode asynchrone et `TypeArguments` est l'ensemble des types paramétrés déclarés sur la méthode appelée. Le label `target` de `MessageSend` a pour cible une expression dont le type est un objet actif, le label `name` a pour cible une chaîne de caractères représentant le nom de la méthode appelée. Les autres labels et nœuds s'expliquent d'eux-mêmes.

La première phase de traitement *back-end* va produire l'AST C en figure 2. Nous avons omis les nœuds représentant des commentaires et les *statements* permettant le *debug*, afin de produire un AST plus lisible. Afin de rendre l'AST plus lisible, nous l'avons également séparé en deux AST, un pour chaque ligne de code Encore. Le véritable AST serait doté d'un nœud `Seq` supplémentaire, avec pour enfants la racine de chacun des deux AST.

Un nœud `Seq` représente une succession d'instructions C (chacune terminée par un point-virgule). Un nœud `Assign` représente une affectation et possède deux enfants : la valeur à affecter, et l'expression à qui l'affecter. Un nœud `Decl` représente une déclaration, et possède deux enfants : le type de la variable et son nom. Un nœud `Call` représente un appel de fonction et possède deux enfants : la fonction à appeler et la liste des paramètres. Les nœuds `AsExpr` sont un détail d'implémentation afin de respecter le système de types d'Haskell, le langage du compilateur.

Enfin, la traduction de cet AST C (seconde phase de traitement *back-end*) produit le code C en listing 7. Nous avons omis les commentaires générés, ainsi que les lignes de code permettant le *debug*, afin de rester proches de l'AST C que nous avons présenté. [7]

7. Les fonction `oneway` en Encore représentent un appel asynchrone où le *future* résultant est ignoré

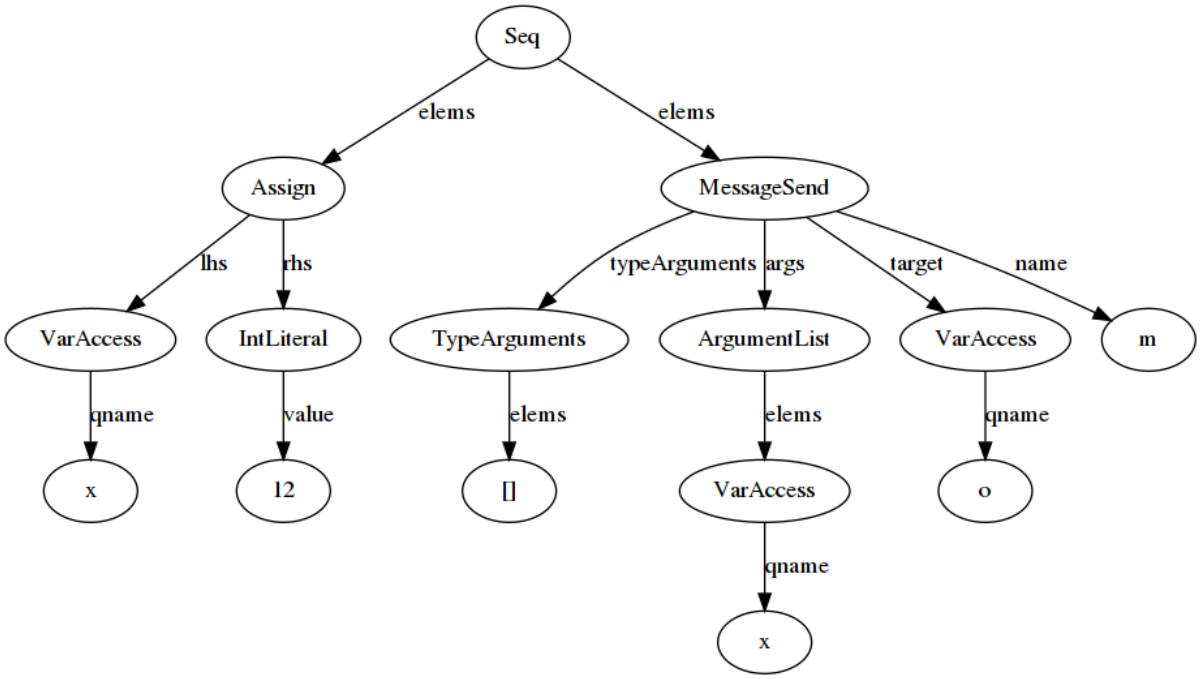
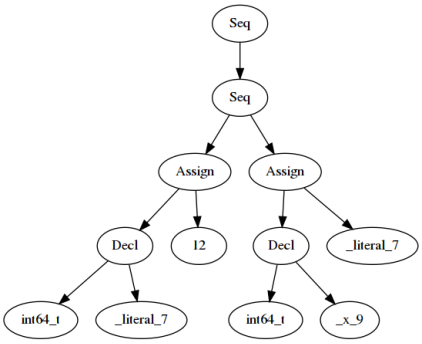
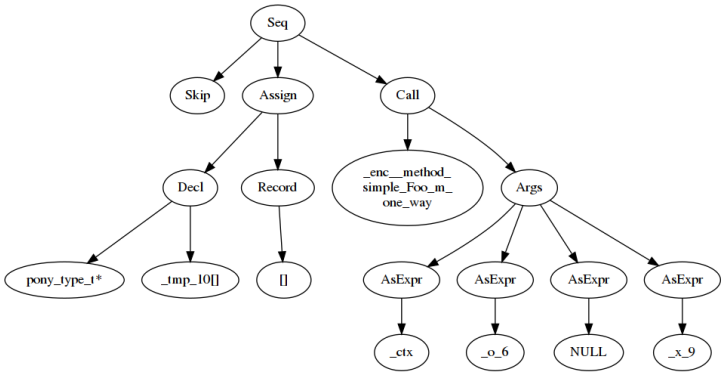


FIGURE 1 – AST Encore issu du code en listing 6



(a) AST C correspondant à la première ligne de code Encore du listing 6



(b) AST C correspondant à la seconde ligne de code Encore du listing 6

FIGURE 2 – AST C après traitement de l'AST Encore en figure 1

Listing 7 – Traduction C du code Encore en listing 6

```
1 int64_t _literal_7 = 12;
2 int64_t _x_9 = _literal_7;
3 pony_type_t* _tmp_10[] = {};
4 _enc__method__simple_Foo_m_one_way(_ctx, _o_6, NULL, _x_9);
```

2.3.3 Généricité dans Encore

Le langage Encore supporte une forme de généricité, c'est-à-dire qu'il est possible d'écrire des fonctions, méthodes et classes de sorte qu'elles puissent travailler sur tous les types possibles. Par exemple, la classe `Array` est capable de travailler indifféremment sur des `int`, des `Fut[int]`, des `bool`...

Syntaxiquement, la généricité notée *`nom`*`[t,u,v]` et peut se présenter au niveau d'un nom de fonction, méthode ou classe. Chacune des lettres entre crochets identifie un type générique, et peut être utilisée là où le compilateur attend un identificateur de type. Par exemple, une fonction `f` générique sur `t`, et prenant en paramètre une valeur de type générique `t` se noterait `fun f[t](x : t)`.

Il existe traditionnellement deux façons de traiter la généricité au niveau du compilateur : une façon que nous appellerons *compile-time expansion* et une façon que nous appellerons « type erasure ». La *compile-time expansion* est le mécanisme utilisé en C++, le *type erasure* est le mécanisme utilisé en Java.

Compile-time expansion Dans un langage où les génériques sont implémentées au travers du mécanisme de *compile-time expansion*, l'idée est la suivante : le développeur propose un patron de classe, méthode ou fonction. Prenons le listing 8. Ce code définit une fonction *template* en C++ permettant la somme de deux valeurs de type générique `T`. Le paramètre *template* `T` indique que la fonction pourra travailler, en théorie, sur n'importe quel type. On pourra ainsi appeler `add` avec des entiers, des pointeurs, des objets...[8] Le processus de compilation est le suivant : quand le compilateur rencontre un appel à `add` avec des paramètres d'un type `U` pour la première fois dans le code, avec `U` un type quelconque, il prend le code de la fonction *template* et en génère une version où toutes les occurrence de `T` sont remplacées par le type `U` et identifie cette fonction comme `addU`[9]. Les occurrences des `add` appelées avec des paramètres de type `U` dans le code sont remplacée par `addU`. Le même mécanisme s'applique sur les méthodes et classes *templates*.

Type erasure Dans un langage où les génériques sont implémentées au travers du mécanisme de *type erasure*, l'idée est la suivante : le développeur propose une classe, méthode ou fonction en indiquant qu'elle pourra travailler sur tous les types possibles. A la différence de la *compile-time expansion* où le développeur offre des patrons, ici le développeur propose des classes, méthodes et fonctions concrètes. Là où la *compile-time expansion* va produire autant de classes, fonctions et méthodes que nécessaire, le *type erasure* va générer une unique classe, fonction ou méthode et s'assurer que les opérations effectuées à l'intérieur sont correctes. Comparez les listings 8 et 9. Dans le listing 8, le code compilera si la fonction `f` est appelée avec exclusivement des entiers, des flottants ou tout type pour lequel l'opérateur `+` est défini. Dans le listing 9, le code ne compilera pas, à moins que l'opérateur `+` ne soit conçu pour travailler sur tous les types imaginables.

Listing 8 – Illustration du mécanisme de *compile-time expansion* des génériques (C++)

```
1 template<typename T>
2 T add(T a, T b) {
3     return a + b;
4 }
```

Listing 9 – Illustration du mécanisme de *type erasure* des génériques (Encore)

```
1 fun add[t](a : t, b : t) : t
2     return a + b
3 end
```

8. L'exemple est purement académique. Ajouter des pointeurs est totalement dénué de sens, tout comme ajouter certains objets entre eux

9. Dans la pratique le nommage est plus complexe

Comparaison La *compile-time expansion* tend à produire plus de code exécutable, étant donné que le patron est dupliqué autant de fois qu'il est utilisé par des types différents. En contrepartie, un patron non utilisé n'est jamais instancié et donc n'occupe aucune place en mémoire. Une classe générique avec le mécanisme de *type erasure* est nécessairement compilée et existe dans l'exécutable généré, même si le développeur ne s'en sert pas. De plus, il est possible, au travers du mécanisme de *type-erasure* de considérer le type générique comme un type à part entière, éventuellement avec des propriétés. Par exemple, en Java, il est possible d'indiquer qu'un paramètre générique `T` dérive d'une classe `C` ; il est alors possible, sur variable de type `T`, d'appeler des méthodes de `C`.

Le langage Encore utilise le mécanisme de *type erasure*. Nous verrons en quoi cela peut poser problème pour l'implémentation de Godot.

2.3.4 Une introduction au *runtime* d'Encore

Le *runtime* d'Encore est écrit en C et utilise le *runtime* du langage Pony afin de gérer les objets actifs. Afin de familiariser le lecteur avec les concepts que nous évoquerons dans notre contribution, nous offrons ici une introduction au fonctionnement du *runtime* d'Encore.

Classes Pour chaque classe Encore, la traduction génère une structure C, dotée de champs représentant le attributs de la classe Encore. Les types Encore sont représentés par les types C ainsi : les types numériques de C représentent les types numériques d'Encore, et les pointeurs permettent de manipuler tous les autres types.

Méthodes Les méthodes sont traduites par des fonctions C prenant en paramètres supplémentaires trois pointeurs : le pointeur vers l'objet courant `this`, les types utilisées pour instancier les types paramétrés, et le *contexte* Pony, qui permet au *framework* de déterminer quel acteur s'exécute, dans quelle pile...

Builtins Les constructions *builtins*, telles que `Fut`, `Maybe`, ou encore `Par`, sont fournies par des bibliothèques écrites directement en C, fournies avec le compilateur.

Structures fonctionnelles (*closures*, *pattern matching*...) Les structures fonctionnelles sont traduites au cas par cas. Par exemple les *closures* sont transformées en structures dotées de champ pour représenter l'environnement clos, et des fonctions C sont générées pour les exécuter.

Envoi de messages Dans le cadre de Godot, l'aspect le plus important du *runtime* Encore est l'envoi de message, qui correspond aux appels asynchrones des objets actifs. Nous considérons comme exemple le listing 10, qui est une version modifiée du listing 6. Un objet actif de type `Foo` est créé, stocké dans la variable `o`. La méthode `m` de cet objet actif est appelée de façon asynchrone, avec en paramètre la valeur `12`. Le résultat de cet appel est stocké dans la variable `f`, qui est donc un *future*.

Listing 10 – Code Encore d'exemple pour le fonctionnement de l'envoi de message, *i.e* l'appel asynchrone

```
1 var o = new Foo()
2 var f = o!m(12)
```

Déclinaison des méthodes Lorsqu'une méthode Encore `m` est compilée, plusieurs fonctions C sont générées. Une fonction C « canonique », qui correspond à une traduction de la méthode telle qu'elle est écrite en Encore. Cette fonction canonique est appelée pour traduire les appels synchrones. Trois autres fonctions sont générées pour représenter les appels asynchrones, les appels *oneway* et les appels `forward`. Lors de la traduction d'un appel asynchrone à `m`, via l'opérateur `!`, c'est la fonction C représentant un appel asynchrone à `m` qui est appelée [10]. Les fonctions C représentant les appels non synchrones ne font pas directement appel à la fonction canonique. Elles envoient des messages dans le *framework* Pony. Ces messages déclencheront un appel à la fonction canonique une fois reçus.

Messages Un message Encore est représenté par une structure C. Cette structure est dotée d'un *future*, qui sera résolu par l'exécution de la méthode. Cette structure contient également les champs nécessaires pour stocker les valeurs des paramètres envoyées à la méthode. Enfin, un message est identifié de façon unique.

Dispatch Pour chaque classe Encore, le compilateur génère une fonction de *dispatch*. Cette fonction reçoit en paramètre un acteur Pony et un message Pony reçu par cet acteur. Le message possède un champ permettant de l'identifier. La fonction de *dispatch* est alors un `switch` sur l'identifiant du message. Chaque `case` effectue ensuite les actions nécessaires. Dans le cas de la réception d'un message de type *future*, l'action est d'appeler la fonction canonique avec les paramètres stockés dans le message, et de résoudre le *future* associé avec la valeur renvoyée par l'appel à la fonction canonique.

Nommage des fonctions Lorsqu'une méthode Encore est traduite en C, le compilateur considère un nom canonique, obtenu à partir du nom de la classe et du nom de la méthode. Ce nom canonique est le nom de la fonction représentant un appel synchrone à la méthode. Les noms des versions *future*, *oneway*... sont formés à partir de ce nom canonique, en lui ajoutant un suffixe.

Génériques Lors de la traduction d'Encore vers C, les variables Encore typées par un type générique sont traduites par des variables C typées `encore_arg_t`. Ce type est une union qui permet de représenter un entier, un flottant, un booléen ou un pointeur. Lorsqu'une valeur de type « type générique » est renvoyée par une fonction, le compilateur se charge de récupérer le bon champ dans l'union. L'utilisation d'une union a comme conséquence la perte de l'information de type à l'exécution. Il n'est donc pas possible, étant donné une instance `encore_arg_t` quelconque, de déterminer quel est le type qu'elle est censée représenter.

2.3.5 Une illustration du fonctionnement du *runtime* d'Encore

La traduction du code en listing 10 donnerait donc le graphe d'appels présenté dans la figure 3. Nous proposons une version simplifiée du code C généré en annexe C.

Site d'appel Le site d'appel se trouve dans une fonction C, `main` par exemple. L'appel asynchrone Encore est traduit par un appel à la fonction C appropriée. Cet appel produit un *future* dans le *runtime* C, nommons le `fut`.

10. Un appel *oneway* est un appel asynchrone où la valeur de retour de l'appel est ignorée. Un appel `forward` est un appel asynchrone *via* la construction `forward`

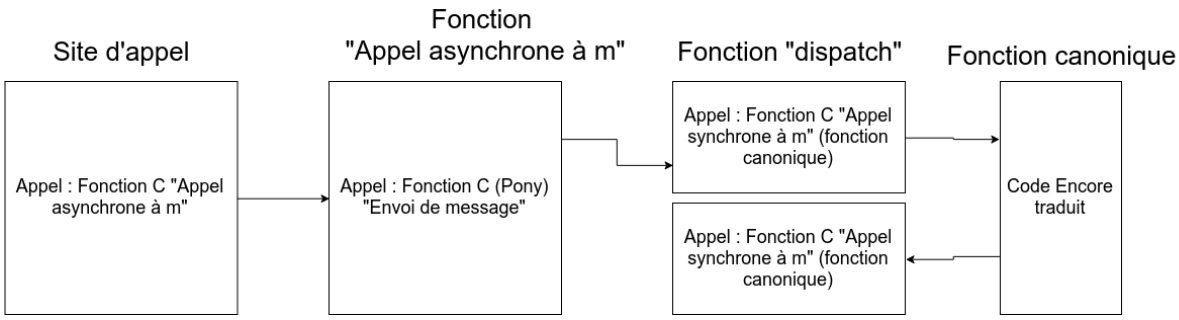


FIGURE 3 – Graphe des appels C déclenchés par un appel asynchrone Encore

Fonction C « Appel asynchrone à `m` » Cette fonction C crée le *future* `fut`, et un message (`msg`). Ce message est initialisé avec un identifiant, et rempli avec le *future* qui sera résolu lors de l'appel effectif à la fonction générée, ainsi qu'avec les paramètres qui seront passés à `m` en C. Enfin, la fonction envoie le message dans le *framework* Pony, puis renvoie `fut` au site d'appel.

Pony Pony envoie ensuite le message à la représentation de l'objet `o` dans le *runtime*. A la réception de ce message, la fonction C de *dispatch* de la classe `Foo` est appelée, avec en paramètres la représentation de `o` et le message `msg`.

Dispatch Grâce à l'identifiant du message, la fonction C de *dispatch* identifie qu'elle doit appeler la fonction C canonique associée à la méthode `m`, et utilise son résultat pour remplir le *future* contenu dans le message.

2.4 Une introduction informelle à Godot

Godot est le système de types proposé par Fernandez-Reyes et al. dans [9]. Nous adoptons leurs notations : `Fut[T]` désigne les *futures* explicites *control-flow* usuels, tels qu'on les trouve dans les langages comme C++ (`future<T> f`) ou Encore (`f : Fut[T]`) ; `Flow[T]` désigne leur nouvelle construction, les *futures* explicites *dataflow*. Par soucis de cohérence, nous adoptons également les termes suivants : un *future* désigne désormais un *future* explicite *control-flow* (`Fut[T]`) ; un *flow* désigne un *future* explicite *dataflow* (`Flow[T]`).

2.4.1 Sous-typage et *collapse*

Le cœur de Godot repose sur un ensemble de règles de typage et sur un opérateur de *collapse*, noté $\downarrow$. Le *collapsing* permet de définir l'idée que le type `Flow` à M niveaux est identique au type `Flow` à N niveaux. L'une des règles de Godot établit que si une variable est de type `T`, alors elle est également de type `Flow[T]`. On dit que `T` est un sous-type de `Flow[T]`.

Sous-typage La règle de sous-typage permet d'utiliser une valeur de type `T` là où une valeur de type `Flow[T]` est attendue. Cette règle permet donc la réutilisation de code que l'on observe dans les *futures* implicites. Si le programmeur souhaite écrire une fonction qui puisse travailler indifféremment sur un `int` ou sur un `Flow[int]`, cette fonction prendra en paramètre un `Flow[int]`. De même, il est possible d'écrire une fonction dont la valeur de retour soit un `Flow[T]` ou un `T`, si la fonction est typée `Flow[T]`. Nous notons cette règle `T` <: `Flow[T]`, lu « `T` sous-type de `Flow[T]` ».

Collapsing Le *collapsing* consiste à appliquer une série de transformations sur un type afin de produire un second type. Par exemple, le type `Flow[Flow[Flow[int]]]` *collapse* en `Flow[int]`.

Notation ↓T La notation ↓`T` désigne le type obtenu après *collapsing* du type `T` ; nous désignons ce nouveau type par « type *collapsé* de `T` ».

- Un type non paramétré *collapse* en lui-même, *e.g* ↓`int` = `int` ;
- Un `Flow[Flow[T]]` *collapse* en ↓`Flow[T]`, *e.g* ↓`Flow[Flow[int]]` = ↓`Flow[int]` ;
- Si `T` n'est pas de la forme `Flow[...]`, `Flow[T]` *collapse* en `Flow[`↓`T]`, *e.g* ↓`Flow[int]` = `Flow[`↓`int]`.
- Un type paramétré `Ty[T, U, ...]`, paramétré par `T`, `U`... *collapse* en `Ty[`↓`T, `↓`U, ...]`, *e.g* ↓`Array[int]` = `Array[`↓`int]` ;

Intérêt du *collapse* Le système de types Godot suppose que tous les types ont subi un *collapse*. L'objectif est de ne jamais créer un type de la forme `Flow[Flow[T]]`. Appeler de façon asynchrone *flow* une fonction renvoyant un `Flow[int]` produit une valeur typée `Flow[int]`, en raison du *collapse*. Il est ainsi possible de typer des chaînes de *flows* non bornées, par exemple celles créées par une factorielle asynchrone. La fonction factorielle asynchrone renverrait une valeur de type `Flow[int]`. L'utilisation du sous-typage permet de typer correctement le cas d'arrêt, qui produit une valeur de type `int`, sous-type de `Flow[int]`. En effectuant un *collapse* sur le type des appels asynchrones à l'intérieur de la factorielle, la fonction renvoie systématiquement une valeur de type `Flow[int]`. Nous traiterons de la factorielle plus en détail dans une future section.

Dans la suite de ce rapport, l'expression « type résultat d'un `Flow[T]` » désignera le type résultat du `Flow` après *collapse*. Par exemple, le type résultat de `Flow[Flow[int]]` est `int`, car ce type *collapse* en `Flow[int]`.

2.4.2 Sémantique de `get*`

Nous présentons ici le fonctionnement de `get*` tel que défini par Godot. Il existe une fonction `isFlow :: a -> Bool`. L'opérateur `::` désigne le type, l'opérateur `->` désigne l'application (une fonction `f` de `int` et `double` vers `bool` serait notée `f :: int -> double -> bool`). `isFlow` prend une valeur de type `a` quelconque et indique si cette valeur est un *flow* ou non. Il est donc nécessaire de pouvoir déterminer à l'exécution si le type d'une variable `v` est *flow* ou non. Nous illustrons le fonctionnement de `get*` par du pseudo-code C dans le listing 11.

Listing 11 – Fonctionnement de `get*` dans Godot

```
1  value_t get*(object_t object)
2  {
3    if (isflow(object))
4    {
5      /* Attend que le flow soit resolu et extrait la valeur de
           resolution */
6      value_t resolved = flow_block((flow_t)flow);
7      return resolved;
8    }
9    else
10   {
11     return (value_t)object;
12   }
13 }
```

Le fonctionnement de `get*` est donc le suivant : si l'objet sur lequel on appelle `get*` est un ↓`Flow[T]`, `get*` attend que le `Flow` soit résolu et renvoie la valeur de résolution. Sinon, `get*` renvoie immédiatement l'objet passé en paramètre. L'absence de récursion dans `get*` est dûe au mécanisme de résolution des `Flow` de Godot. Un `Flow` n'est jamais résolu avec un `Flow`. Considérons une chaîne de 3 `Flow`, créée par une succession de trois appels asynchrones. Lorsque le troisième et dernier `Flow` est résolu avec une valeur `v`, un chaînage est attaché au second `Flow` pour le résoudre avec `v`. Lorsque le second `Flow` est résolu avec `v`, un chaînage est attaché au premier `Flow` pour le résoudre avec `v`. C'est ainsi que `get*` peut produire une valeur non `Flow` en une seule attente.

2.4.3 Remarques

Il est à noter que Godot ne fournit que peu d'éléments d'implémentation de sa sémantique. Comment est-ce que les conversions de `T` vers `Flow[T]` ou de `Flow[Flow[T]]` vers `Flow[T]` sont effectuées n'est pas précisé. Il n'est pas non plus indiqué comment est-ce que `get*` peut être implémenté dans un langage qui ne propose pas de mécanisme d'introspection natif. L'implémentation de la fonction `isFlow` est donc à la convenance du développeur.

Godot présente également une sémantique pour les opérations `then*`, `forward*` et un encodage des *futures* à partir de *flow*. Tout cela échappe à notre contribution aussi nous n'en parlerons pas.

3 Une implémentation des *dataflow explicite futures* sans introspection native

Nous présentons à présent notre contribution : une implémentation de Godot dans le langage Encore, et le processus de compilation associé. En particulier, nous insistons sur la solution que nous apportons au problème suivant : comment, dans un langage dépourvu d'introspection, fournir une implémentation de Godot ?

Dans un premier temps, nous considérons que le langage Encore est dépourvu de fonctions, méthodes ou classes génériques. La généricité introduit plusieurs problèmes, que nous traitons en section 3.3 et en section 3.4.

3.1 Implémentation des *dataflow explicit futures* dans Encore

Nous invitons le lecteur à se référer à la section 2.3.1 s'il souhaite se familiariser avec les notations et opérations sur `Fut` dans Encore afin de pouvoir y comparer nos choix de notations et d'opérations sur `Flow`.

Afin d'illustrer notre contribution, nous prenons comme exemple la fonction factorielle [11]. Le listing 12 présente le code qu'un utilisateur souhaite être capable d'écrire. Notez qu'il doit également être possible d'écrire ce code à l'aide d'`async*` et sans classes (nous présentons ce même code écrit à l'aide d'`async*` en annexe, dans le listing 24).

Nous pouvons observer les particularités suivantes sur ce listing 12 :

- Le programme utilise une construction `Flow[T]` ;
- Cette construction suit la règle de *collapsing* de Godot : avec les *futures* explicites *control-flow*, la méthode `fact_acc` ne serait pas typable. Ici, elle est typée `Flow[int]` ;

11. Nous avons choisi cette fonction car il s'agit d'une fonction récursive simple, qui permet d'illustrer l'utilisation d'appels récursifs asynchrones

Listing 12 – Factorielle asynchrone avec `Flow` dans Encore

```
1  import Task
2
3  active class Fact
4      def fact(n : int) : Flow[int]
5          return this!!fact_acc(1, n)
6      end
7
8      def fact_acc(r : int, n : int) : Flow[int]
9          if (n <= 0) then
10             return r
11         else
12             return this!!fact_acc(r * n, n - 1)
13         end
14     end
15 end
16
17 active class Main
18     def main() : unit
19         println(get*(new Fact()!!fact(10))) -- 3628800
20     end
21 end
```

- L'expression `return r` à la ligne 10 est proprement typée. Le compilateur traite, à minima, le type `int` comme sous-type de `Flow[int]`. Le *runtime* dispose d'un outil de conversion de la représentation de `int` vers la représentation de `Flow[int]` ;
- Le programme utilise une fonction `get*`, prenant en paramètre un `Flow[int]` et renvoyant une valeur de type `int` ;
- Le programme utilise un opérateur `!!` permettant d'appeler une méthode de façon asynchrone dans un contexte *flow*, de manière analogue à l'opérateur `!` pour un contexte *future*.

Ce code présente ce qu'un développeur souhaiterait pouvoir faire dans le langage Encore : créer des `Flow` qui suivent les règles de Godot, et manipuler ces `Flow`. Nous allons présenter ce que nous avons mis en place dans le compilateur pour implémenter cela : quelles constructions nous avons ajoutées, comment nous les avons traduites, et comment est-ce que nous avons résolu le problème de l'absence d'introspection dans le *runtime* d'Encore.

3.1.1 Syntaxe

Nous présentons les ajouts que nous avons apportés à la syntaxe d'Encore : la construction `Flow[T]`, l'opérateur `!!` et la fonction `get*`. Ces modifications se font dans le *parser*, dans le système de types et dans l'AST.

Parser Nous avons ajouté la construction `Flow` dans le *parser*, en tant que type paramétré. Nous avons également ajouté l'opérateur `!!` comme opérateur binaire, et `get*` [12] comme mot-clé.

12. La structure du *parser* rend délicat le traitement de `get*` en tant que fonction, étant donné que le caractère `*` est réservé comme opérateur multiplicatif et que les noms de fonctions ne peuvent pas contenir de caractères spéciaux

Système de types Nous avons ajouté un type (`FlowType`) à l'ensemble des types que connaît le compilateur. Cet ajout nous permettra d'annoter correctement l'AST pour permettre le *typechecking*.

AST Nous avons ajouté de nouveaux types de nœuds à l'AST pour représenter ces constructions. Le nœud `GetStar` représente une utilisation de l'instruction `get*` et contient l'expression sur laquelle la fonction est appelée. Le nœud `MessageSendFlow` représente une utilisation de l'opérateur `!!` et contient l'expression à gauche de l'opérateur, le nom de la méthode appelée, les paramètres de la méthode appelée et des informations sur les types des paramètres génériques.

3.1.2 Typage

Nous présentons ici les modifications apportées au *typechecking*, afin de permettre la mise en place du *collapsing* et de la règle de sous-typage `T <: Flow[T]` [13].

Sous-typage Le compilateur d'Encore propose la fonction Haskell `subtypeOf`. Cette fonction prend en paramètres deux types Encore, et indique si le premier est sous-type du second. Nous avons ajouté le cas suivant à cette fonction pour mettre en place le sous-typage de Godot. Nous considérons que tous les types avec lesquels nous allons travailler dans cette explication sont *collapsés*. Soient `T` et `U` les types Encore passés en paramètres. Si `U` est de la forme `Flow[T']`, `T` est sous-type de `U` si `T` est sous-type de `T'`.

Au *runtime* La traduction du code Encore en code C créera des situations où des valeurs de type `int` seront passées à des valeurs de type *flow* dans le *runtime*. Nous dotons le *runtime* d'une fonction permettant de créer un *flow* à partir d'une valeur. Nous créons également un nouveau type de nœud (`LiftToFlow`). Dans l'AST, ces nœuds contiendront une expression. Lors de la traduction, un tel nœud sera traduit par un appel à la fonction C de création d'un *flow* depuis une valeur.

Collapsing Nous avons défini la fonction `collapseFlow` qui prend en paramètre un type `T` et le *collapse* en lui appliquant l'opérateur ↓ récursivement. Nous proposons son code dans le listing 13. La fonction distingue trois cas :

- Le type que l'on veut *collapse* est un type à résultat *builtin* (`Fut`, `Maybe`...), de la forme `T[R]`, avec `R` le type résultant. `T[R]` *collapse* en `T[↓R]` ;
- Le type que l'on veut *collapse* est une classe paramétrée, de la forme `T[U, V...]`, avec `U`, `V`... les types paramètres. `T[U, V...]` *collapse* en `C[↓U, ↓V, ↓...]` ;
- Le cas par défaut. Aucun changement n'est à effectuer.

Dans le *typechecking* Lors de la phase de *precheck*, durant laquelle l'AST est annoté avec les types pour permettre le *typechecking*, nous avons *collapsé* les types : chaque fois qu'un nœud doit être annoté avec un type `T`, nous l'annotons avec `↓T` (grâce à la fonction `collapseFlow`). Cela permet de ne pas avoir à effectuer de *collapse* à d'autres points du code, et de manipuler des types `Flow` à un seul niveau.

13. Par abus de langage, nous utiliserons souvent l'expression « type Encore », ou un type Encore directement, en place de « la représentation dans le compilateur du type Encore »

Listing 13 – Implémentation de la fonction `collapseFlow` en Haskell

```haskell
-- Collapse les Flow dans un type parametre, recursivement
-- collapseFlow Flow[Flow[int]] -> Flow[int]
-- collapseFlow Array[Flow[Flow[Array[Flow[Flow[int]]]]]] -> Array[Flow
   [Array[Flow[int]]]]
collapseFlow :: Type -> Type
collapseFlow ty
  | hasResultType ty =
      let collapsed = collapseFlow $ getResultType ty
      in setResultType ty collapsed
  | isClassType ty =
      let parameters = getClassParameters ty
          collapsed  = map collapseFlow parameters
      in setClassParameters ty collapsed
  | otherwise = return ty
```

Conversions Lors du *typechecking* de nœuds d'affectation (passage de paramètre, utilisation de l'opérateur `=`, instruction `return`), nous effectuons une vérification sur le type utilisé pour annoter chacun de ces nœuds. De tels nœuds manipulent nécessairement deux expressions : la cible, *i.e* ce qui reçoit l'affectation, et la source, *i.e* la valeur que l'on affecte. Dans le cas où la cible est de type ↓`Flow[T]` et où la source est de type ↓`T`, la traduction devra intégrer l'appel, au *runtime*, à une fonction de conversion. A cette fin, nous remplaçons le nœud « source » de l'affectation dans l'AST par le nœud de conversion (`LiftToFlow`). L'expression stockée dans ce nœud de conversion est l'ancien nœud « source » de l'affectation. Enfin, nous annotons le nœud de conversion avec le type ↓`Flow[T]`. Le listing 14 illustre un cas où une telle conversion est requise. Les figures 4 et 5 montrent, respectivement, les AST Encore associés à ce code avant et après transformation.

Listing 14 – Illustration d'un cas de *lift* en Encore

```
fun f() : Flow[int]
    return 42
end
```

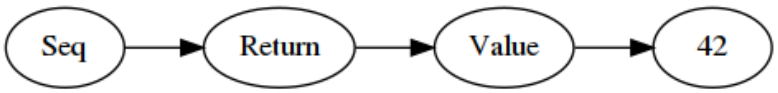


FIGURE 4 – AST Encore correspondant au listing 14 avant *lift*

FIGURE 5 – AST Encore correspondant au listing 14 après *lift*

3.1.3 *Runtime* et génération de code

La dernière étape consiste en l'implémentation en `C` de la bibliothèque de gestion des *flows*, et en la génération du code.

Comme nous l'avons vu en section 2.3.2, la génération de code se fait en deux étapes : une traduction de l'AST Encore vers un AST C, puis un parcours de cet AST C afin de générer le code C associé. Étant donné que les nœuds de l'AST C couvrent l'ensemble des structures syntaxiques du C, il n'est pas nécessaire d'y toucher. La compilation va donc se concentrer sur la traduction de l'AST Encore vers l'AST C.

Si le lecteur désire se familiariser avec le fonctionnement du *runtime* d'Encore, et en particulier avec le fonctionnement des échanges de messages, nous le renvoyons vers la section 2.3.4.

Nous présentons dans cette section le travail que nous avons effectué au niveau de la compilation des `Flow` : comment nous avons représenté les `Flow` dans le *runtime*, comment nous avons mis en place les appels asynchrones dans le *runtime*, et, en particulier, comment nous avons mis en place l'introspection de Godot dans le *runtime* pour permettre la synchronisation.

Le fonctionnement des appels asynchrones *flows* est similaire au fonctionnement des appels asynchrones *futures*. Appeler une méthode ou fonction de façon asynchrone *flow* crée un *flow* dans le *runtime* et déclenche un envoi de message. La réception de ce message déclenche un appel à la fonction C appropriée. La valeur renvoyée par cette fonction est utilisée pour résoudre le *flow* initialement créé.

La mise en place de la gestion des appels asynchrones *flow* au *runtime* est une adaptation directe de la gestion des appels asynchrones *futures*. Nous présentons succinctement ce travail d'adaptation par souci d'exhaustivité, mais il ne s'agit en rien de notre principale contribution. Le système utilisé pour les appels *futures* est opérationnel, et notre objectif n'était pas de l'améliorer, c'est pourquoi nous nous sommes contentés d'une adaptation. Nous renvoyons le lecteur vers la section 2.3.4 s'il souhaite une introduction au *runtime* d'Encore.

`Flow` dans le *runtime* Nous représentons le type Encore `Flow` par une structure C (`flow_t`). Nous avons doté cette structure d'un champ de type `encore_arg_t` qui représente la valeur avec laquelle le `Flow` Encore associé sera résolu. La structure contient également un champ booléen indiquant si le `Flow` associé est résolu.

Création de *flows* Un *flow* peut être résolu par une valeur immédiate, en raison de la règle de sous-typage. Nous pouvons donc créer un `flow_t` de deux façons en C : en le remplissant immédiatement, ou en attendant pour le remplir par le résultat d'un appel asynchrone. Nous avons créé deux fonctions :

- La fonction `flow_mk` permet de créer une instance de `flow_t` où le champ résultat n'est pas initialisé, et où le booléen indiquant la résolution est à `false` ;
- La fonction `flow_mk_from_value` qui permet de créer une instance de `flow_t` dans laquelle le champ résultat est immédiatement initialisé avec une valeur, et où le booléen indiquant la résolution est à `true`. Les nœuds de conversion de `↓T` vers `↓Flow[T]` de l'AST sont traduits, chacun, par un appel à cette fonction. Cette fonction n'est utilisée qu'à des fins de conversions, et n'est **jamais** utilisée pour créer un `Flow` immédiatement résolu par un autre `Flow`[14]. Nous nommons la conversion d'une valeur en un `flow_t` un « *lift* ».

Résolution de *flow* Afin de permettre la résolution d'une instance de `Flow`, nous avons créé une fonction C dédiée (`flow_fulfil`). Cette fonction permet de donner une valeur au champ résultat d'une instance de `flow_t`. Elle est utilisée à la réception d'un message asynchrone pour résoudre une instance de `flow_t`.

Appels asynchrones *flow* Comme nous l'avons vu précédemment, un appel asynchrone *flow* se comporte de façon presque identique à un appel asynchrone *future*. Nous avons donc reproduit le processus de traitement des appels asynchrones *futures* en remplaçant les éléments propres aux *futures* par leur équivalent *flow*.

14. Un tel cas est absurde par construction. Un `Flow` résout un autre `Flow` uniquement dans le cas d'appels asynchrones enchaînés. Une telle configuration ne requiert pas de résolution immédiate.

Fonction d'envoi de message Lors de la traduction d'une méthode Encore `m`, nous créons une fonction C supplémentaire. Cette nouvelle fonction C représentera un appel asynchrone *flow* à `m`.

Messages De nouvelles structures de messages sont créées, avec un `flow_t` en attribut (le *flow* qui sera résolu à la réception du message). De nouveaux identifiants sont créés pour identifier les instances de ces nouvelles structures de messages.

Corps des fonctions « appel asynchrone *flow* » Le corps de ces fonctions est une adaptation à *flow* du corps des fonctions « appel asynchrone *future* ». Elles créent et renvoient un `flow_t` au lieu de `future_t` afin de traiter le bon type. Au lieu d'invoquer `future_mk`, elles invoquent `flow_mk`. Au lieu d'envoyer des messages *future*, elles envoient des messages *flow*.

Dispatch Afin de traiter les messages *flow* nous ajoutons de nouveaux cas dans la fonction de *dispatch*, similaires aux appels *futures* qu'ils complètent. Dans chacun de ces cas, au lieu d'invoquer la fonction de résolution des *futures*, la fonction invoque la fonction de résolution des *flows*. Cet appel à la fonction de résolution des *flows* prend en paramètre le résultat de l'appel à la fonction canonique.

3.1.4 Introspection & Synchronisation

La fonction Encore `get*` permet de déclencher une synchronisation *dataflow* sur une instance de `Flow`. Comme nous l'avons vu dans notre présentation de Godot (section 2.4), la sémantique de `get*` fait intervenir un outil d'introspection. Dans Godot, cet outil est une fonction nommée `isFlow` qui détermine, à l'exécution, si une valeur quelconque est de type `Flow` ou non. Toutefois, le *runtime* d'Encore est dépourvu d'introspection. Nous présentons ici notre analyse de ce problème, et la solution que nous avons apportée. Nous continuons de travailler exclusivement avec des types qui ont subi le *collapse*.

Par abus de langage et pour le confort du lecteur, nous emploierons l'expression « l'exécution du code Encore » en place de « l'exécution de la traduction en C du code Encore ». Par ailleurs, nous raisonnerons essentiellement sur des méthodes Encore. Le raisonnement se généralise aux fonctions Encore sans changements. Enfin, toujours dans un souci de confort de lecture, nous utiliserons la notation *flow* pour désigner la construction `flow_t` dans le *runtime*.

Rappel du problème Dans le *back-end* du compilateur, tous les types `Flow` ont subi un *collapse*. Il n'est donc pas possible de déterminer statiquement combien de synchronisations seront nécessaires sur un ↓`Flow[T]` pour obtenir une valeur de type ↓`T`. Un ↓`Flow[T]` peut représenter un `Flow` à un, deux, trois, ou n niveaux.

Exemple de la factorielle Considérons notre factorielle en listing 12. Au *runtime*, l'exécution de `new Fact()!!fact(10)` va créer un total de 12 instances de *flow* : 10 pour la récursion, une pour l'appel à `fact_acc` et une pour l'appel à `fact`. Nous pouvons affirmer que précisément 12 instances seront créées car nous avons déroulé le code à la main. Nous pouvons observer que ces instances de *flow* présentent une relation de dépendance : chaque nouvelle instance créée par un appel sera utilisée comme valeur de résolution de l'instance créée par l'appel précédent. L'instance de *flow* créée par appel asynchrone à `fact_acc(1, 10)` résoudra l'instance de *flow* créée par appel asynchrone à `fact(10)`. Une fois toutes les instances créés, une structure chaînée apparaît : chaque instance, à l'exception de la douzième (la dernière), est résolue par une instance de *flow* différente. Nous pouvons donc traverser la chaîne en utilisant le champ résultat sur les

onze premières instances. Le champ résultat de la dernière instance est rempli avec le résultat du calcul de la factorielle de 10, qui est une valeur (telle que définie en introduction).

Analyse de l'exemple La seule instance de `flow_t` dont le champ résultat ne réfère pas à une autre instance de *flow* est l'instance résolue dès sa création, à savoir la douzième. En effet, la traduction de la ligne 10 du listing 12 est un appel à la fonction `flow_mk_from_value` : la fonction Encore renvoie un `Flow[int]`, le `return` produit un `int`, une conversion est donc nécessaire au *runtime*. Par définition, une instance de *flow* créée par `flow_mk_from_value` contient un résultat qui n'est pas une instance *flow*. Nous avons un premier élément utilisable pour identifier si une instance de *flow* est résolue avec une valeur ou avec une autre instance de *flow*.

Étude des types de retour de méthodes Nous avons vu qu'un `flow_t` créé par la fonction C `flow_mk_from_value` est toujours résolu par une valeur. Nous allons à présent étudier la fonction `flow_mk`. Nous allons considérer deux méthodes Encore. Les appels asynchrones à ces deux méthodes créeront, à l'exécution, des instances de *flow* à l'aide de la fonction C `flow_mk`.

- Considérons ce premier en-tête de méthode Encore : `def f(): int`. Si nous appelons cette méthode de façon asynchrone *flow*, nous obtenons un résultat typé `Flow[int]`. Nous savons, par définition que, dans le `flow_t` représentant ce `Flow[int]`, le champ résultat contiendra une valeur de type `int`.
- Considérons ce second en-tête de méthode Encore : `def g(): Flow[int]`. Si nous appelons cette méthode de façon asynchrone *flow*, nous obtenons un résultat typé `Flow[int]`. Nous ne pouvons pas savoir combien de `Flow` précisément seront créés à l'exécution, mais nous pouvons affirmer qu'il y en aura au moins deux. Le premier sera issu de l'appel asynchrone à `g`. Le second sera celui renvoyé par `g`. Enfin, nous savons que le `Flow` renvoyé par `g` résoudra le `Flow` créé par l'appel asynchrone.

Analyse Nous pouvons observer que le type de retour d'une méthode `m` permet de déterminer si un appel asynchrone à `m` produira une instance de *flow* résolue par une valeur ou par une autre instance de *flow*. Si le type de retour est de la forme ↓`Flow[T]`, l'instance de *flow* créée à l'exécution sera résolue par une autre instance de *flow*. Si le type de retour n'est pas de la forme ↓`Flow[T]`, l'instance de *flow* créée à l'exécution sera résolue par une valeur.

Synthèse Nous avons donc un moyen de déterminer si une instance de *flow* est destinée à être résolue par une autre instance de *flow* ou par une valeur. Nous proposons l'algorithme suivant :

- Si une instance de *flow* est créée par un appel à `flow_mk_from_value`, elle est résolue par une valeur ;
- Si une instance de *flow* est créée par un appel à `flow_mk`, nous distinguons deux cas :
  - Si la fonction appelée de façon asynchrone possède un type de retour de la forme ↓`Flow[T]`, l'instance de *flow* est résolue par une autre instance de *flow* ;
  - Sinon, l'instance de *flow* est résolue par une valeur.

Introspection à l'exécution Nous devons propager cette information à l'exécution, afin de permettre une implémentation de `get*`. L'introspection supposée par Godot permet de déterminer à l'exécution si une valeur quelconque est de type `Flow` ou non. Nous proposons une autre forme d'introspection : chaque instance de *flow* est capable de fournir l'information « Est-ce que mon champ résultat contient / est destiné à contenir une instance de *flow* ou une valeur ». Pour ce faire, nous dotons la structure `flow_t` d'un champ supplémentaire (`result_type`) donnant cette information. Nous modifions le corps de la fonction `flow_mk_from_value` afin de configurer cette information. Le comportement de la fonction `flow_mk` dépend du contexte, aussi nous la

dotons d'un paramètre supplémentaire qui indiquera si le *flow* créé doit être résolu par une valeur ou par une autre instance de *flow*. Nous modifions également le compilateur afin de fournir cette information dans les appels à `flow_mk`.

Écriture de l'information à la compilation Le compilateur génère des appels à `flow_mk` exclusivement dans le corps des fonctions C « appel asynchrone *flow* à une méthode Encore ». Lorsque ces fonctions sont générées, le compilateur évalue le type de retour de la méthode en cours de traduction. Si le type de retour est de la forme ↓`Flow[T]`, l'appel à `flow_mk` prend en paramètre une valeur indiquant que le *flow* résultant sera résolu par une autre instance de *flow*. Sinon, l'appel prend en paramètre une valeur indiquant que le *flow* sera résolu par une valeur.

Comparaison avec Godot Godot suppose l'existence d'une fonction `isFlow :: t -> Bool`, qui indique pour toute valeur si celle-ci est un *flow* (au sens de Godot) ou non. Considérons le pseudo-code suivant : `flow_t x = 3`. D'après Godot, `(isFlow x)` == `true`, et `(isFlow 3)` == `false`. Notre implémentation ne propose ni la fonction `isFlow`, ni les mêmes informations. Dans Godot, toute valeur peut être inspectée pour déterminer s'il s'agit d'un `Flow` ou non. Dans notre implémentation, seules les instances de `flow_t` peuvent être inspectées, et indiquent si leur *résultat* est un `flow_t` ou non. Si nous implémentons `isFlow :: flow_t -> Bool`, et que nous reprenons le pseudo-code précédent, dans notre implémentation `(isFlow x)` == `false` et `(isFlow 3)` n'est pas correctement typé.

Avantages et limitations Notre solution limite l'ajout de l'introspection aux entités qui en ont besoin. Nous plaçons l'information uniquement dans les *flow*, contrairement à Godot. En contrepartie, comme nous le verrons en section 3.3, notre forme d'introspection est mise à mal par le fonctionnement de la généricité d'Encore. L'introspection supposée par Godot ne serait pas concernée par ces problèmes.

Une implémentation de `get*` Nous traduisons la fonction Encore `get*` par une nouvelle fonction C (`flow_get`). L'implémentation de cette fonction est présentée au listing 15. La fonction `flow_get` attend que le `flow_t` en paramètre soit résolu. Après cela, elle évalue la valeur du champ indiquant le type du résultat (`result_type`) : si cette valeur n'est pas `FLOW`, `flow_get` s'interrompt immédiatement en renvoyant la valeur de résolution ; sinon, `flow_get` *cast* le résultat du *flow* en un `flow_t*` et s'appelle récursivement sur ce `flow_t*`.

Comparaison avec Godot En section 2.4, nous avons présenté le fonctionnement de `get*` dans Godot. En particulier, nous avons montré que Godot ne propose pas un `get*` récursif ; Godot empêche un `Flow` d'être résolu avec un autre `Flow` en utilisant du chaînage. `get*` peut alors fournir une valeur utilisable après une seule attente au maximum. Dans Godot, le chaînage est possible car il existe un état global qui indique quel *flow* est attaché à quelle tâche, et quel valeur est produite par quelle tâche. Dans le *runtime* d'Encore, cet état global n'existe pas, du moins pas sous une forme aisément manipulable. C'est pourquoi nous avons choisi de mettre en place `get*` au travers d'une récursion. L'approche de Godot évite un empilement d'appels, au prix de la création de plusieurs tâches asynchrones afin d'exécuter le chaînage. Notre approche évite la création de nouvelles tâches asynchrones, au prix d'un empilement d'appels, mais est plus adapté à notre contexte.

Ainsi, nous avons une implémentation de `Flow` dans Encore et son *runtime*. En particulier, nous avons trouvé une solution à l'absence d'introspection dans le *runtime* d'Encore en étudiant

Listing 15 – Implémentation de `get*` en C

```
encore_arg_t flow_get(pony_ctx_t** ctx, flow_t* flow)
{
  if (!flow->fulfiled) {
      flow_block_actor(ctx, flow);
  }

  assert(flow->fulfiled);
  acquire_flow_value(ctx, flow);

  if (flow->result_type == FLOW) {
    return flow_get(ctx, (flow_t*)flow->result.p);
  } else {
    return flow->result;
  }
}
```

la résolution des `Flow`, ce qui nous a permis d'ajouter une forme d'introspection, différente de celle attendue par Godot, mais offrant le même résultat (en excluant le traitement des génériques).

Nous avons omis de parler de l'opérateur `async*` et des appels asynchrones de fonctions globales. Par soucis de complétion, nous présentons à présent notre travail d'implémentation de ces deux fonctionnalités.

3.2 Le cas d'`async*` et des fonctions globales

Nous avons ajouté la construction `async*` dans le *parser* en tant qu'opérateur unaire, de façon similaire à l'opérateur `async`. Notre implémentation de l'opérateur est similaire à celle d'`async` (qui permet d'invoquer une fonction globale de façon asynchrone *future*). Tout comme `async` *desugar* en un appel à la fonction globale Encore `spawn`, `async*` *desugar* en un appel à la fonction globale Encore `spawn_star` que nous avons codée. La fonction `spawn_star` utilise l'opérateur `!!` sur un objet actif pour lancer la fonction de façon asynchrone. Nous présentons le code de la fonction `spawn` en listing 17 et le code de la fonction `spawn_star` en listing 16.

Listing 16 – Implémentation de `spawn_star` en Encore

```
fun spawn_star[t](task : () -> t) : Flow[t]
  let runner = new TaskRunner()
  in runner!!perform(task)
end
```

Listing 17 – Implémentation de `spawn` en Encore

```
fun spawn[t](task : () -> t) : Fut[t]
  let runner = new TaskRunner()
  in runner!perform(task)
end
```

Listing 18 – Implémentation du `TaskRunner` en Encore

```
active class TaskRunner
  def perform[t](task : () -> t) : t
    task()
  end
end
```

La conversion de l'expression passée en paramètre à `async*` en une fonction *lambda* permet de faire fonctionner `spawn_star` pour n'importe quel appel de fonction globale. En effet, s'il devait exister une version de `spawn_star` pour chaque type de fonction, avec tous les types de paramètres possibles, il faudrait une infinité de versions de `spawn_star` différentes. En transformant l'expression d'async* en une continuation, typée `() -> t` (une fonction ne prenant pas de

paramètres et renvoyant une valeur de type générique `t`), il est possible de typer `spawn_star :: (() -> t) -> Flow[t]`, pour tous les types `t` possibles, à l'aide des génériques.

L'obtention d'un *future* (pour `spawn`) ou d'un *flow* (pour `spawn_star`) se fait au moyen d'un appel asynchrone *future* (dans le cas de `spawn`) ou *flow* (dans le cas de `spawn_star`). Il est ainsi possible de réutiliser le code du TaskRunner (listing 18) qui était déjà utilisé pour `spawn`, en remplaçant l'opérateur `!` dans l'appel à `perform` par l'opérateur `!!`. Le `TaskRunner` est un objet actif utilisé uniquement dans le but de pouvoir lancer des appels asynchrones *futures* ou *flows* depuis les fonctions `spawn` et `spawn_star`. Cela permet de ne pas avoir à effectuer une implémentation particulière pour les opérateurs `async` et `async*` : leur implémentation repose sur l'utilisation des objets actifs.

L'implémentation d'`async*` conclut nos travaux de compilation des `Flow`. Nous allons à présent traiter des fonctions, méthodes et classes génériques qui présentent plusieurs problématiques, en raison de leur implémentation dans le compilateur. Nous présentons ces problèmes et les solutions que nous avons explorées afin de les résoudre.

3.3 Le cas des génériques

Nous invitons le lecteur à se référer à la section 2.3.3 s'il souhaite se familiariser avec la généricité, aussi bien de façon générale qu'au sein d'Encore.

Nous avons présenté en section 3.1.4 un algorithme permettant de déterminer si une instance de `flow_t` sera résolue avec une autre instance de `flow_t` ou avec une valeur. Nous avons également présenté comment conserver cette information à l'exécution, en ajoutant un champ à la structure `flow_t`. Appeler de façon asynchrone une fonction ou méthode Encore ne renvoyant pas un `Flow`, ou effectuer un *lift* crée une instance de `flow_t` qui sera résolue avec une valeur. Toutes les autres créations de `Flow` résultent en des créations d'instances de `flow_t` résolues avec d'autres instances de `flow_t`.

Le cœur de cet algorithme repose sur la capacité du compilateur à déterminer si une méthode ou fonction renvoie une valeur de type *flow* ou non. Pour cela, nous avons créé une fonction Haskell (`isFlowType`) indiquant si une valeur est de type statique `Flow` ou non.

3.3.1 Les génériques dans le compilateur

Considérons l'en-tête d'une méthode générique Encore : `def f[t](x : t): t`. Cette méthode prend en paramètre une valeur de type générique `t` et renvoie une valeur de type générique `t`. Nous avons précédemment (section 2.3.3) remarqué qu'Encore utilise le mécanisme de *type-erasure* pour l'implémentation de ses génériques : chaque classe, méthode et fonction générique est traduite une seule fois, et le *typechecker* s'assure que le typage est correct pour tous les types possibles. Nous pouvons donc nous demander « Quel est le type du paramètre `x` dans `f` ? Quel est le type de retour de `f` ? ». Ces questions se posent également dans le cas des fonctions génériques.

En interne, le compilateur d'Encore dispose d'un type dédié pour représenter les types génériques : `TypeVar`. Dans l'AST, le paramètre `x` et le type de retour de la méthode `f` sont annotés avec le type `TypeVar`. Par conséquent, la fonction Haskell `isFlowType` appliquée sur le type de retour de `f`, ou sur son paramètre `x`, renverra faux. Selon notre algorithme, appeler `f` de façon asynchrone *flow* produira un *flow* destiné à être résolu par une valeur. Cette conclusion fournie par l'algorithme est vraie si le paramètre générique de `f` dans l'appel n'est pas $\downarrow$`Flow`, mais est fausse dans les autres cas.

Nous présentons les solutions que nous avons envisagées pour résoudre ce problème. Dans la suite de cette section, nous nous concentrerons sur les méthodes génériques exclusivement. Elles présentent plusieurs problématiques qui leurs sont propres. Nous traiterons les fonctions et classes génériques en section 3.3.3.

3.3.2 Solutions envisagées

Notre problématique demeure liée au besoin de déterminer, à l'exécution, avec quel type est-ce qu'une instance de *flow* est résolue. Notre précédent algorithme reste valide, à condition de le restreindre aux méthodes non génériques. Quelle que soit la solution que nous envisageons, elle reposera sur le même principe : fournir une information, à l'exécution, sur le type de valeurs.

Introspection booléenne en C La première solution que nous avons envisagée consiste à transformer l'union `encore_arg_t` en une structure. Cette structure serait dotée d'une union, afin de conserver la possibilité de représenter tous les types Encore en C, et d'un champ booléen. Le booléen indiquerait si la valeur contenue dans l'union est un *flow* ou non.

Configurer le booléen Un `encore_arg_t` contient un `flow_t` si et seulement s'il est créé à partir du résultat d'un appel à l'une des fonctions permettant de créer des `flow_t` (les fonctions C `flow_mk` et `flow_mk_from_value`, voir section 3.1.3). Le booléen est donc placé à `true` dans ces cas-là, et est par défaut à `false`.

Avantages et inconvénients Cette solution a l'avantage de fonctionner facilement quel que soit le nombre de paramètres génériques de la méthode, ce qui ne sera pas le cas d'autres solutions que nous avons envisagées. En contre-partie, l'utilisateur doit payer le coût de l'introspection même lorsqu'il n'utilise pas de `Flow` dans le code Encore. En effet, l'union `encore_arg_t` est utilisée par de nombreuses constructions du langage, telles que les génériques de façon générale. L'un des objectifs du *runtime* C d'Encore est d'être efficace, et le langage Encore respecte le principe de « si l'utilisateur n'utilise pas tel outil, alors il n'en paye pas le coût d'utilisation ». Notre approche ajouterait le coût d'un booléen pour chaque instance d'`encore_arg_t`, ce qui va à l'encontre de ce principe. De plus, cette solution demande une réécriture massive des bibliothèques C d'Encore, en raison du passage d'une union à une structure, ainsi que des modules standards d'Encore qui peuvent faire des appels directs à du code C. Enfin, il serait nécessaire de changer toute la génération de code, y compris celle non liée aux *flows*, pour traiter le passage d'une union à une structure.

Compile-time expansion La seconde solution que nous avons envisagée consiste à effectuer une multiplication des fonctions C générées pour traduire les méthodes Encore. Pour une méthode générique, dotée de N paramètres génériques, nous générerions 2^N fonction C. Chacune de ces fonctions représenterait une configuration des paramètres génériques : aucun n'est `Flow`, le premier seul est `Flow`, tous sont `Flow`... Afin de couvrir toutes les façons possibles d'invoquer une méthode Encore (*future*, *flow*, *oneway*...), nous multiplierions également les fonctions C correspondant à ces formes d'invocation. Soit M le nombre de façons différentes d'invoquer une méthode Encore. Nous générerions au final $M \times 2^N$ fonctions C.

Déterminer quelle fonction C appeler Pour simplifier la discussion, considérons le cas d'un appel synchrone à une méthode (le processus est identique pour toutes les autres formes d'appels). Afin de déterminer quelle version synchrone appeler, nous devons déterminer ce que les paramètres génériques de la méthode deviennent dans cet appel. Nous effectuons cela en

évaluant les paramètres passés à la méthode lors de l'appel. Nous présentons en listing 20 un algorithme permettant cette évaluation.

Avantages et inconvénients Cette solution a l'avantage de ne pas créer de surcoût inutile à l'exécution, moyennant que l'éditeur de liens supprime les versions qui ne sont jamais appelées. Une optimisation possible à la compilation consisterait à évaluer le graphe d'appels de méthodes pour déterminer lesquelles dupliquer ou non. En contre-partie, sur des programmes qui manipulent intensivement génériques et `Flow` il est possible de se retrouver avec de nombreuses versions C d'une même méthode Encore. En effet, si une méthode Encore possède deux paramètres génériques, il faut générer quatre versions *flow* de cette méthode pour chaque procédé d'invocation. Enfin, le corps de toutes les variations des fonctions C serait similaire, seuls changeraient les identifiants de messages ou les noms des fonctions C appelées par la suite. Cette duplication de code n'est pas optimale, mais peut être optimisée s'il s'avère que cette solution est intéressante.

Introspection booléenne localisée La troisième solution que nous avons envisagée consiste à combiner les deux approches précédentes. Au lieu de multiplier les traductions des méthodes Encore en C, nous ajoutons un paramètre supplémentaire aux fonctions C correspondant aux traductions des méthodes Encore et aux invocations asynchrones des méthodes Encore : un tableau de booléens. Le i-ème booléen de ce tableau indique si le i-ème paramètre générique est un `Flow` ou non dans l'appel Encore.

Fonctionnement Le fonctionnement est basé sur celui de la méthode *compile-time expansion*. Tout comme dans l'*expansion* brute, il est nécessaire de déterminer quel paramètre générique devient `Flow` dans chaque appel de méthode Encore. La traduction de l'appel est modifiée pour recevoir le tableau de booléens en plus. Ce tableau doit être envoyé dans les messages échangés par les objets actifs afin que les méthodes appelées de façon asynchrones sachent quels paramètres génériques sont `Flow` ou non.

Avantages et inconvénients Cette solution présente l'avantage de ne pas multiplier abusivement les fonctions C, contrairement à l'*expansion* brute. Le surcoût est également localisé, contrairement à l'ajout d'un booléen brut dans `encore_arg_t`. En contrepartie, il est nécessaire de payer un léger surcoût, et le corps des fonctions C devient plus complexe, étant donné qu'il faut traiter le tableau de booléens.

Utilisation de mécanismes natifs pour implémenter la solution En section 2.3.4, nous avons indiqué que les fonctions C associées aux méthodes et fonctions Encore prennent en paramètre un tableau d'identifiants des types utilisés pour instancier les paramètres génériques dans chaque appel. *A priori*, nous pourrions penser que cela résout notre problème. Nous sommes toutefois confrontés à des problèmes d'implémentation. Nous avons choisi de traiter le type `Flow` comme un type *builtin* dans le compilateur, car un `Flow` n'est ni un objet actif, ni un acteur. Il s'agit d'un type obéissant à ses propres règles en matière de gestion de la mémoire et des interactions possibles avec, au même titre que `Fut` ou `Maybe`. À l'exécution, les types *builtins* ne sont pas identifiés individuellement : ils partagent tous un même identifiant (`PRIMITIVE`). Changer cet identifiant, ou ne plus traiter le type `Flow` comme un *builtin*, conduirait le *runtime* à le traiter comme un acteur, ce qui pose de nombreux problèmes de gestion de la mémoire. Utiliser ce tableau d'identifiants ne résout donc pas notre problème.

3.3.3 Fonctions et classes génériques

Nous présentons succinctement comment est-ce que chacune des solutions que nous avons envisagées s'appliquent aux fonctions et classes génériques.

Fonctions génériques Les fonctions sont systématiquement appelées de façon synchrones. En effet, appeler une fonction de façon asynchrone revient à l'appeler de façon synchrone dans un objet actif créé à la volée. L'ajout d'un booléen sur toutes les instances d'`encore_arg_t` permet de traiter nativement les fonctions génériques. La solution de *compile-time expansion* se limite à générer 2^N fonctions C pour traduire une fonction Encore, avec N le nombre de paramètres génériques. Nous utilisons le même algorithme d'évaluation du type d'un paramètre générique présenté en listing 20 afin de déterminer quelle version appelée. Enfin, la solution d'introspection booléenne localisée se contente d'ajouter le tableau de booléen en paramètre à la fonction, et la logique de traitement du tableau dans le corps de la fonction C.

Classes génériques Les classes Encore sont traduites par une structure C, et un ensemble de fonctions C travaillant sur cette structure. L'ajout d'un booléen dans toutes les instances d'`encore_arg_t` permet de traiter nativement les classes génériques. Dans les deux autres solutions, dans la représentation C de chaque classe générique Encore, nous ajoutons un tableau de booléens indiquant quels paramètres génériques sont `Flow` ou non dans cette instance. Nous identifions quels paramètres sont `Flow` ou non à l'aide de l'algorithme d'évaluation de type présenté en listing 20. Lorsqu'une méthode Encore utilise un paramètre générique issu de la classe, le tableau de booléens est consulté afin de déterminer si ce paramètre est un `Flow` ou non.

3.4 *Compile-time expansion* des génériques

Nous avons retenu la seconde solution : multiplier les traductions des méthodes génériques nécessaires, et invoquer la bonne version selon les types utilisés pour instancier les paramètres génériques dans chaque appel. Nous avons choisi cette solution pour deux raisons : la première est l'absence de surcoût inutile à l'exécution ; la seconde est la limitation de la quantité de code à modifier : nous modifions exclusivement le code du compilateur [15].

Dans notre mise en œuvre, nous nous sommes limités aux méthodes et fonctions génériques dotées d'un seul paramètre générique. Ce cas est plus simple à traiter et, une fois traité, se généralise. Si une méthode générique possède un paramètre générique, et se situe dans une classe dotée d'un paramètre générique, nous considérons que la méthode possède deux paramètres génériques ; par conséquent nous ne traitons pas ce cas.

Dans cette section nous nous focaliserons exclusivement sur les méthodes génériques, pour deux raisons. Premièrement, les appels de fonctions génériques sont traités de la même façon que les appels synchrones de méthodes [16]. Deuxièmement, les classes génériques utilisent un processus qui est une sous-partie du processus utilisé sur les méthodes génériques. Notre travail sur les méthodes génériques est donc applicable aux fonctions et classes génériques sans modifications majeures.

Un exemple simple Considérons un exemple illustrant le cas le plus simple de notre problème dans le listing 19. Nous étudierons un cas plus complexe par la suite.

15. Une troisième raison est que nous n'avons pas envisagé la solution d'introspection booléenne localisée avant de commencer notre implémentation

16. Un appel asynchrone à une fonction correspond à un appel synchrone à la fonction dans un objet actif créé à la volée.

Listing 19 – Utilisation de `Flow` avec les génériques

```
1  active class Foo
2      def foo[t](x : t) : t
3          x
4      end
5
6      def bar[t](x : t) : t
7          this.foo(x)
8      end
9  end
10
11 active class Main
12     def main() : unit
13         var f = new Foo()
14         var fl = f!!foo(12) -- Flow[int]
15         var fl2 = f!!foo(fl) -- Flow[Flow[int]]
16         var fl3 = f!!bar(12) -- Flow[int]
17     end
18 end
```

Présentation et analyse de l'exemple Le listing 19 présente une classe active `Foo` dotée de deux méthodes génériques `foo` et `bar`. `foo` est la fonction identité, `bar` se contente d'appeler `foo` de façon synchrone. La méthode `main` de la classe `Main` crée trois `Flow` : deux par appel à `foo`, un par appel à `bar`. Le premier et le troisième appel ne posent pas de problème. Intéressons-nous au second appel.

Résultat attendu La ligne 15 est traduite par un appel à la fonction d'invocation asynchrone *flow* C de la méthode `foo`. Avant modification du compilateur, on trouve dans le corps de cette fonction C un appel à `flow_mk` ; le *flow* créé par cet appel sera paramétré comme « résolu par une valeur ». En effet, le type statique de retour de la méthode `foo` est « type générique », qui n'est, **statiquement**, pas le type `Flow`. Nous souhaitons que la ligne 15 soit traduite par un appel C à la fonction d'invocation asynchrone *flow* de la méthode `foo`, instanciée avec `Flow`. Dans le corps de cette nouvelle fonction C, on trouvera un appel à la fonction `flow_mk` ; le *flow* créé par cet appel sera paramétré comme « résolu par un *flow* ».

3.4.1 Solution retenue

Nous utiliserons la formulation « version *procédé* instanciée sur $t_1, t_2, t_3 \ldots$ de la méthode *méthode* » pour identifier les fonctions C. *procédé* indiquera quelle méthode d'invocation nous utiliserons (*flow*, *future*...). Les t_i prendront les valeurs `Flow` ou `Other` pour indiquer si le i-ème paramètre générique de la méthode est instancié avec `Flow` ou non. Par exemple « Version *flow* instanciée sur `Flow`, `Other` de la méthode `foo` » désigne la fonction C représentant un appel asynchrone *flow* en Encore à la méthode `foo`. Dans cet appel Encore, le premier paramètre générique de `foo` est `Flow`, le second est n'importe quel autre type distinct de `Flow`.

Génération de nouvelles fonctions La solution repose sur une déclinaison des fonctions C. Nous pouvons rapprocher cela de la spécialisation des *templates* de C++. Lorsqu'une méthode Encore est traduite, les fonctions C représentant les invocations à cette méthode sont dupliquées. Nous avons ainsi la version synchrone instanciée avec `Flow`, la version *future* instanciée avec `Flow`... en plus de la version synchrone instanciée avec `Other`, de la version *future* instanciée

avec `Other`... Pour simplifier la discussion, nous considérons dans la suite que tous les appels sont synchrones (le raisonnement est identique pour tous les procédés d'invocation).

Identification de la version à appeler Étant donné que nous ne traitons que le cas où les méthodes génériques Encore ont un unique paramètre générique, identifier la version C à appeler est plus simple que dans le cas général. Nous pouvons identifier quatre cas d'appels de méthodes :

- Appeler une méthode non générique depuis une méthode non générique ; ce cas ne pose aucun problème . Ce cas est déjà fonctionnel, grâce au processus que nous avons décrit dans la section 3.1.4 ;
- Appeler une méthode générique depuis une méthode non générique ; nous pouvons déterminer immédiatement si le type générique devient `Flow` dans cet appel car tous les types des paramètres sont connus ;
- Appeler une méthode non générique depuis une méthode générique ; en raison du fonctionnement des génériques d'Encore, ce cas ne présente pas de problèmes. Si la méthode non générique attend une valeur de type `Flow` en paramètre, lui passer une valeur de type `t` sera rejeté par le *typechecker*. En d'autres termes, les paramètres génériques n'ont aucune influence sur la traduction d'un appel de méthode non générique ;
- Appeler une méthode générique depuis une méthode générique ; nous devons déterminer si le paramètre générique de la méthode appelée devient `Flow` dans cet appel. Ce processus se fait en plusieurs étapes, aussi nous allons le détailler.

Appel d'une générique depuis une générique Nous rappelons que nous traitons exclusivement les méthodes génériques à un seul paramètre générique. Nommons `src` la méthode appelante, `dst` la méthode appelée, `t` le paramètre générique de `src` et `u` le paramètre générique de `dst`. Nous pouvons identifier deux cas :

- Une valeur de type générique `t` est passée en paramètre à `dst`[17]. Si nous sommes en train de générer la version instanciée avec `Flow` de `src`, nous devons effectuer un appel à la version instanciée avec `Flow` de `dst`. Sinon nous appelons la version instanciée avec `Other` ;
- Aucune valeur de type générique `t` n'est passée en paramètre à `dst`. Nous déterminons si `u` devient `Flow` dans cet appel, au moyen de l'algorithme d'évaluation de type présenté en listing 20. Si `u` devient `Flow`, nous appelons la version instanciée avec `Flow` de `dst`. Autrement, nous appelons la version instanciée avec `Other`.

3.4.2 Implémentation

Dans un premier temps, nous nous concentrons sur les modifications apportées au code C. Nous considérons désormais et les appels synchrones et les appels non synchrones des méthodes Encore, en raison de variations dans le traitement.

Nommage des nouvelles fonctions C Nous commençons par définir les noms des fonctions C qui représentent la version instanciée avec `Flow` d'une méthode générique Encore. Nous effectuons du *name mangling*, similaire à ce que l'on trouve dans les compilateurs C++ pour permettre la surcharge de méthodes[18].

17. Par soucis de simplicité de l'explication nous ne mentionnons que le cas où la méthode appelée attend un `t` en paramètre. Nous serons plus exhaustif lorsque nous étudierons le processus dans le détail.

18. L'idée est de « décorer » le nom de la fonction, par exemple en lui ajoutant un suffixe, afin de différencier quelle fonction travaille sur `Other` et quelle fonction travaille sur `Flow`

Corps des nouvelles fonctions C Le corps de chacune de ces nouvelles fonctions est très similaire au code la fonction d'origine qu'elle duplique. Dans le cas de la fonction C représentant un appel synchrone, seuls les appels à d'autres méthodes Encore sont susceptibles de changer. Chaque appel de méthode générique est traduit par un appel à la version instanciée sur `Flow` ou sur `Other` selon les cas. Nous expliquerons plus loin comment nous déterminons cela. Dans le cas des fonctions C représentant les appels asynchrones, l'identifiant de message et la structure du message utilisés seront différents, afin de les distinguer de leurs versions instanciées sur `Other`. En effet, l'appel Encore `o!m(x)` avec `m` une générique, peut désormais être traduit de deux façons en C, en fonction du type utilisé pour instancier le paramètre générique.

Duplication des messages Appeler une méthode Encore de façon asynchrone s'effectue en deux temps dans le code C. Premièrement, une fonction C est invoquée et envoie un message. Deuxièmement, à la réception du message, la fonction C associée à la méthode est invoquée. Si le paramètre de la méthode générique devient `Flow` dans un appel asynchrone, il faut appeler la fonction C instanciée sur `Flow` à la réception du message. Nous dupliquons donc les structures de messages en C associés aux appels asynchrones de méthodes génériques (voir section 2.3.4). Nous dupliquons également les identifiants des messages. Tout comme pour les noms de fonctions C, nous effectuons du *name mangling* afin de distinguer la version `Flow` et la version `Other` des structures de messages et de leurs identifiants. Enfin, afin de traiter ces nouveaux messages, nous générons des cas additionnels dans la fonction de *dispatch*.

Fonctions de création d'objets Tout objet Encore dispose d'une méthode implicite `init` qui fait office de constructeur. Lorsque nous créons la structure C associée à une classe générique Encore dotée d'un seul paramètre, nous ajoutons un champ booléen à la structure indiquant si le paramètre générique de cette instance de la classe Encore est de type `Flow`.

Nous présentons à présent les modifications que nous avons apportées au compilateur.

Duplication Lors de la traduction d'une méthode générique (ou d'une méthode non générique dans une classe générique), nous évaluons son nombre de paramètres génériques, ainsi que le nombre de paramètres génériques de sa classe. Si la somme des deux vaut 1, nous traduisons deux fois la méthode : une version instanciée sur `Flow`, une version instanciée sur `Other`. Dans le cas des fonctions globales nous évaluons le nombre de paramètres génériques et traduisons deux fois si ce nombre est 1.

Traduction des appels de méthodes et fonctions Lorsque nous traduisons un appel de méthode, nous évaluons à nouveau le nombre de paramètres génériques de cette méthode et de sa classe. Si ce nombre est 1, nous lançons l'algorithme présenté en listing 20 permettant de déterminer s'il faut traduire l'appel Encore par un appel C à la fonction instanciée avec `Flow` ou non. Nous procédons de façon similaire pour les fonctions génériques.

Traduction des créations d'objets Lorsque nous traduisons une création d'objet, nous évaluons le nombre de paramètres génériques de la classe de l'objet. Si ce nombre est 1, nous lançons notre algorithme pour déterminer s'il faut traduire l'appel Encore par un appel C à la fonction de création instanciée avec `Flow` ou non.

Algorithme d'évaluation de types Nous présentons dans le listing 20 notre algorithme en Haskell. Cet algorithme est utilisé chaque fois que nous devons traduire un appel à une méthode ou fonction Encore générique. La valeur renvoyée indique si la traduction à cet appel

de fonction doit être un appel à la fonction C instanciée sur `Flow`. Nous utilisons deux fonctions Haskell pour cela : `requiresSpecialization` et `checkBindingsForFlow`. La vérification du nombre de paramètres génériques est omise.

Fonction `requiresSpecialization` Cette fonction détermine si un appel de fonction / méthode Encore doit être traduit par un appel à la version instanciée sur `Flow` de la fonction C associée. Cette fonction reçoit quatre paramètres. Les trois premiers sont transférés à `checkBindingsForFlow`, nous les expliquerons dans le prochain paragraphe. Le paramètre `targetTemplate` indique si la fonction appelée est générique ou non. Si oui, il est nécessaire de vérifier si son paramètre générique effectif est un `Flow` dans cet appel. Si non, il n'est pas nécessaire d'appeler la version instanciée sur `Flow` de la fonction C associée.

Fonction `checkBindingsForFlow` Cette fonction détermine si le paramètre générique d'une méthode / fonction Encore devient `Flow` dans un appel donné. Elle reçoit trois paramètres :

- `header` est un en-tête de fonction / méthode. Dans l'utilisation de l'algorithme, il s'agit de l'en-tête de la fonction / méthode appelée ;
- `args` est une liste d'expressions. Dans l'utilisation de l'algorithme, il s'agit des arguments passés à la fonction / méthode appelée ;
- `flowCtx` est un booléen. Dans l'utilisation de l'algorithme, il indique si nous sommes en train de traduire la version instanciée sur `Flow` de la fonction / méthode appelante.

Afin de déterminer si le paramètre générique devient `Flow` dans l'appel, il est nécessaire d'identifier les différents cas où il peut le devenir. Nommons ce paramètre `t`. Nous avons identifié trois cas :

- Un paramètre de la fonction / méthode appelée est de type statique `t`. Par exemple `fun f[t](x : t): t` ;
- Un paramètre de la fonction / méthode appelée est de type statique `T[t]`, avec `T` un type paramétré. Par exemple `fun f[t](x : Fut[t]): t` ;
- Un paramètre de la fonction / méthode appelée est une continuation dont l'un des paramètres est de type statique `t`. Par exemple `fun f[t](x : int -> t): t`.
  - Ces deux derniers cas (type paramétré et continuation) doivent être traités récursivement. Un type paramétré peut être paramétré par un autre type paramétré ou par une continuation. Une continuation peut prendre en paramètres un type paramétré ou une autre continuation. Par exemple `fun f[t](x : int -> Fut[t]): int` ou encore `fun f[t](x : Fut[int -> t]): int`.

La fonction `checkBindingsForFlow` procède ensuite ainsi :

1. Elle construit le symbole `params` qui s'évalue en la liste des types des paramètres de l'en-tête de fonction / méthode ;
2. Elle construit le symbole `bindings` qui s'évalue en la liste des paires (type du paramètre, type de la valeur passée) ;
3. Elle appelle la fonction standard Haskell `any` avec en paramètres une fonction unaire `checkOneBindingForFlow` et le symbole `bindings`. La fonction `any` prend en paramètre un prédicat unaire, et une liste. `any` applique ensuite le prédicat sur chaque élément de la liste. Si le prédicat renvoie `true` sur un élément, `any` renvoie immédiatement `true`. Sinon, `any` renvoie `false`. La fonction `checkOneBindingForFlow` se charge de déterminer si le paramètre générique de la fonction / méthode appelée est instancié avec `Flow` dans cet appel. Une explication complète de cette fonction se trouve en annexe E.

Listing 20 – Algorithme permettant de déterminer si un paramètre générique devient `Flow`

```
checkBindingsForFlow :: A.FunctionHeader -> A.Arguments -> Bool -> Bool
checkBindingsForFlow header args flowCtx =
  let
    params = map (A.ptype) $ A.hparams header
    bindings = zip params $ (map A.getType) args
  in
    any checkBindingForFlow bindings

requiresSpecialization :: A.FunctionHeader -> A.Arguments -> Bool ->
                          Bool -> Bool
requiresSpecialization header args flowCtx targetTemplate =
  if not targetTemplate then
    False
  else
    checkBindingsForFlow header args flowCtx
```

3.4.3 Généralisation

Bien que nous n'ayons pas eu le temps de généraliser notre travail aux fonctions et méthodes génériques dotées de plus d'un paramètre générique, nous présentons ici nos réflexions et analyses sur le procédé.

Nous sommes confrontés aux mêmes problèmes que dans le cas à un seul paramètre générique : dupliquer et distinguer fonctions C, structures et identifiants de messages, et ajouter les cas appropriés dans la fonction de *dispatch*. Enfin, aspect le plus crucial, nous devons généraliser l'algorithme précédent afin de déterminer quelle version de chaque fonction C appeler lors de la traduction d'un appel Encore.

Classes Aux structures de classes génériques nous ajoutons N booléens pour représenter les N paramètres génériques. Le i-ème booléen prend la valeur `true` pour indiquer que le i-ème paramètre générique est `Flow`, `false` autrement.

Corps des fonctions Tous les appels de méthodes ou de fonctions sont remplacés par les appels aux versions appropriées en fonction des types utilisés pour instancier les paramètres génériques. Les fonctions de créations d'objets placent les booléens de structure de classe aux bonnes valeurs. Les fonctions d'envois de messages utilisent les bons identifiants et les bonnes structures de messages.

Duplication des méthodes et fonctions Encore Nous généralisons le processus de traduction. Lors de la traduction d'une méthode Encore, nous générons $M \times 2^N$ fonctions C pour représenter toutes les combinaisons possibles des paramètres génériques (`Flow` ou `Other`), pour chaque façon d'invoquer la méthode. M est le nombre de façons d'invoquer une méthode, N est la somme du nombre de paramètres génériques de la méthode et de sa classe. Dans le cas des fonctions génériques, nous générons 2^N fonctions C pour représenter toutes les combinaisons possibles des paramètres génériques (`Flow` ou `Other`). N représente le nombre de paramètres génériques de la fonction Encore.

Traduction des appels de méthodes et fonctions Lors de la traduction d'un appel de fonction ou de méthode, nous faisons tourner notre algorithme sur chaque paramètre générique

de la fonction / méthode (et sa classe) appelée. Le résultat est un ensemble de booléens qui indique quelle version de la fonction C associée appeler.

Traduction des créations d'objets Lors de la traduction d'une création d'objet, nous faisons tourner notre algorithme sur chacun des paramètres génériques de la classe. Le résultat est un ensemble de booléens qui permettront de remplir les booléens C dans la structure C associée à la classe.

Algorithme Nous n'avons pas mis en place une version généralisée de l'algorithme. A priori, nous estimons qu'il n'est pas nécessaire de le modifier intensivement. Au lieu de prendre en paramètre un booléen indiquant si nous sommes dans la version instanciée avec `Flow` d'une fonction C, il prendrait désormais un tableau de booléens. La modification la plus importante serait de déterminer quel paramètre générique de la méthode appelante devient quel paramètre générique de la méthode appelée. Dans tous les cas, il n'est pas nécessaire de faire travailler l'algorithme sur tous les paramètres génériques de la méthode appelée, nous pouvons l'appliquer paramètre par paramètre. Haskell se prête facilement à ce genre d'opérations, notamment avec `map`. Il nous faut toutefois effectuer une mise en pratique pour confirmer nos hypothèses.

4 Travaux liés

A notre connaissance, il n'existe à ce jour aucune autre étude d'une implémentation de Godot dans un langage de programmation. Nous avons cependant identifié plusieurs travaux qui en reproduisent certains principes.

Fernandez-Reyes et al. se sont intéressés au problème de la délégation : comment une tâche asynchrone peut-elle déléguer la résolution de son *future* à une autre tâche ? Ils proposent la construction `forward` dans [8], aujourd'hui implémentée dans le langage Encore. `forward` visait à réduire le nombre de *futures* à l'exécution dans les langages à objets actifs, *i.e.* Une conséquence est la limiation des constructions `Fut[Fut[T]]`, en explicitant un *collapse*. Godot apporte cet avantage de `forward` en proposant un *collapse* automatique des `Flow`, tout en évitant au programmeur l'utilisation d'une construction dédiée.

Fernandez-Reyes et al. ont également proposé la construction `ParT` dans [10], également implémentée dans le langage Encore. `ParT` est un type paramétré sur `T`. Une instance de `Par[T]` représente une collection de calculs parallèles produisant des valeurs de type `T`, avec lesquels il est possible d'interagir. L'objectif principal `ParT` ne présente aucune similarité avec le fonctionnement de Godot ; c'est un point spécifique de son implémentation qui nous intéresse. Un `Par[T]` peut contenir des valeurs de type `T`, Fut[T] ou encore `Fut[Par[T]]`, et se comporte différemment selon le type de la valeur. Il existe une opération nommée *lift* permettant d'ajouter une valeur dans un `Par[T]`. Fernandez-Reyes et al. ont rencontré un problème similaire à notre implémentation : comment *lift* une valeur de type `t` ? L'absence d'information statique a poussé les développeurs à proposer, en Encore, trois déclinaisons de *lift* : *liftv* permettant de *lift* une valeur non *future*, *liftf* permettant de *lift* une valeur *future* et *liftfp* permettant de *lift* une valeur de type *future* sur `ParT`. L'inconvénient de cette approche est que le développeur doit lui-même effectuer des duplications de fonctions et méthodes génériques Encore selon qu'elles sont destinées à traiter des valeurs non *futures*, *futures* ou *futures* sur `ParT`. Nous avons refusé cette alternative. Notre réponse à la perte d'information statique résout ce besoin de duplication manuelle en effectuant la duplication au moment de la compilation. Cette duplication est possible car nous traquons l'information de type à travers les appels, selon la version de la fonction que nous générons. Une forme plus générale de notre algorithme permettant de déterminer si un

paramètre générique devient un `Flow` pourrait permettre la mise en place d'une fonction *lift* unique dans `ParT`.

Dans un précédent travail d'étude bibliographique nous avons présenté une implémentation brute de la fonction `get*` de Godot dans le langage Python. Contrairement au langage Encore, le langage Python dispose d'outils d'introspection à l'exécution assez puissants pour permettre une implémentation presque immédiate de l'opérateur `isFlow` de Godot. Nos présents travaux apportent toutefois une analyse plus poussée de l'importance de l'introspection dans Godot, ainsi que plusieurs pistes pour mettre en place Godot dans un langage dépourvu d'introspection à l'exécution.

Enfin, Fernandez-Reyes a effectué une mise en œuvre partielle de Godot dans le langage Scala[2]. Son implémentation fonctionne si l'on exclut les types génériques. Lorsque des types génériques paramétrés par *flow* apparaissent, le compilateur n'effectue pas de *collapse*, ce qui lève des erreurs de typage. Notre implémentation évite ce problème en forçant un *collapse*, y compris dans l'évaluation des types génériques. Fernandez-Reyes a choisi de ne pas modifier le compilateur Scala. Son implémentation du *collapse* repose sur un ajout de règles de conversion implicites. Cette restriction dans l'approche a l'avantage de ne pas être intrusive, et d'être, ainsi, adaptée à un compilateur aussi riche que Scala. L'implémentation obtenue est cependant moins générale que ce nous proposons ici, et notre approche permet de compiler un plus grand nombre d'exemples d'utilisation de *flow*.

5 Conclusion

Le système de calcul Godot a mis en évidence une construction pour des *dataflow explicite futures*. Nous avons étudié la possibilité de mettre en place cette construction dans le langage à objets actifs Encore. Notre tentative se conclut par une mise en place opérationnelle de Godot sur un sous-ensemble des fonctions et méthodes du langage Encore : celles dénuées de généricité.

La généricité met en avant l'importance de l'introspection dans toute implémentation de Godot. Comme nous l'avons vu, des problématiques liées aux génériques ont déjà été rencontrées dans la mise en place de la construction *ParT* dans le langage Encore. C'est là l'un des axes de travail les plus importants d'une mise en place de Godot.

Nous avons proposé trois solutions à ce problème, et avons évalué la possibilité de mettre l'une d'entre elles en place. Effectuer un processus similaire à l'instanciation des *templates* de C++ permet de résoudre les problèmes posés par les génériques et l'absence d'introspection dans le *runtime* d'Encore. Le prix à payer est toutefois une explosion combinatoire du nombre de fonctions générées lors de la compilation. Bien que nous ayons choisi cet axe de travail, une solution légère plus coûteuse à l'exécution, mais ne requérant pas de duplication de code, serait l'utilisation de ce que nous avons appelé l'introspection booléenne localisée. En transportant des tableaux de booléens à travers les fonctions C pour déterminer si les paramètres génériques Encore sont *flow* ou non, nous pourrions supprimer l'explosion combinatoire, en la remplaçant par des passages tableaux de booléens entre fonctions. Cette approche semble plus prometteuse dans le cas général, mais légèrement plus difficile à mettre en œuvre que la preuve de concept réalisée durant ce stage.

Un point soulevé par une comparaison des problématiques rencontrées par Fernandez-Reyes dans son implémentation en Scala, et notre propre expérience avec le compilateur d'Encore, est l'importance d'un accès au compilateur pour l'implémentation de Godot. Fernandez-Reyes a obtenu très rapidement des résultats probants en l'absence de génériques, sans toucher au compilateur ; toutefois, l'absence de *collapse* systématique a provoqué des problèmes à posteriori, problèmes qui ne pourraient être corrigés qu'en modifiant le compilateur de Scala. Quant à nous, un accès au compilateur d'Encore a certes permis une mise en place plus flexible, mais au prix

d'une longue étude préliminaire du compilateur.

Notre travail sur la généricité nous a permis de fournir une implémentation de Godot sur les fonctions et méthodes non génériques, ainsi que sur les fonctions et méthodes génériques dotées d'un seul paramètre générique. A terme, il serait intéressant de généraliser notre travail sur la généricité afin de couvrir toutes les fonctions et méthodes génériques ; l'approche la plus prometteuse semble celle où nous remplaçons la multiplication des fonctions C par l'utilisation d'un tableau de booléen. Nous avons pu tester notre travail accompli sur plusieurs programmes d'exemple, et le résultat est fonctionnel.

Références

[1] The encore compiler. `http://github.com/parapluu/encore`. Accessed : 2019-07-23.

[2] Godot : All the benefits of implicit and explicit futures (artefact). `https://github.com/kikofernandez/ecoop-artefact-godot`. Accessed : 2019-08-26.

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

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

[8] 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.

[9] Kiko Fernandez-Reyes, Dave Clarke, Ludovic Henrio, Einar Broch Johnsen, and Tobias Wrigstad. Godot : All the Benefits of Implicit and Explicit Futures. In Alastair F. Donaldson, editor, *33rd European Conference on Object-Oriented Programming (ECOOP 2019)*, volume 134 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 2 :1–2 :28, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

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

Annexes

A Les *futures* comme *framework* d'échange de messages

Les *futures* sont une construction intéressante en programmation asynchrone, car ils permettent de mettre en place une forme de communication par messages. Nous pouvons représenter le lancement d'une tâche asynchrone par un envoi de message. Lorsqu'une tâche produit une valeur, nous pouvons le représenter par un envoi de message implicite. Récupérer la valeur contenue dans un *future* s'apparente alors à une réception de message.

Nous pouvons comparer l'utilisation des *futures* à l'utilisation de *frameworks* de plus bas niveau tels que MPI. Avec les *futures* les envois et réceptions de messages sont plus aisément garantis. Nous comparons le travail nécessaire selon chaque approche dans la table 1.

	MPI		*Futures*	
Action	Processus 1	Processus 2	Processus 1	Processus 2
Lancement	`MPI_Send`	`MPI_Recv` + Appel	`async`	
Récupération	`MPI_Recv`	`MPI_Send`		`return`

TABLE 1 – Comparaison du lancement de tâche asynchrone et réception de valeur entre MPI et les *futures*

Lancer une tâche avec MPI demande au programmeur d'effectuer un `MPI_Send` dans un processus et d'effectuer un `MPI_Recv` dans un autre processus. Oublier l'un ou l'autre empêchera l'application de fonctionner correctement. De plus, le message envoyé au second processus devra identifier l'action à effectuer à la réception comme un lancement de tâche. Pour lancer une tâche avec les *futures*, le programmeur appelle `async` et indique le nom de la tâche à exécuter, dans le premier processus. Le second processus n'a rien à faire.

Récupérer la valeur calculée par la tâche avec MPI demande au programmeur d'effectuer un `MPI_Send` dans le second processus et un `MPI_Recv` dans le premier processus. Oublier l'un ou l'autre empêchera l'application de fonctionner correctement. Tout comme pour le lancement de tâche, le message devra s'identifier comme indicateur d'une valeur calculée. Pour envoyer une valeur au premier processus avec les *futures*, le programmeur utilise un `return`. Cette approche permet de s'abstraire du concept de *future* en codant ses tâches comme des fonctions, et une analyse statique peut aisément détecter si un chemin de contrôle ne produit pas de valeur.

Bien qu'en apparence les *futures* semblent une solution idéale comparée à un outil plus bas niveau tel que MPI, ils n'en sont pas un remplacement. Un *future* représente avant tout une valeur future, en cours de calcul, et non un échange de message. Dans un langage acteur, où les entités peuvent être réparties sur plusieurs machines différentes, la communication entre les machines peut se faire via MPI, qui est un outil bien plus approprié. Les *futures* sont alors une abstraction au-dessus de MPI afin de simplifier le travail du programmeur.

B *Futures* implicites et explicites

Les *futures* sont traditionnellement répartis en deux catégories : les *futures* implicites et les *futures* explicites. Cette répartition peut s'appliquer sur chacun des trois aspects suivant : création, accès et typage. Par exemple, nous pouvons imaginer des *futures* explicites par création, et implicites par accès et typage. Diverses combinaisons sont possibles et présentent chacune avantages et inconvénients.

Création Un *future* est explicite par création s'il existe une instruction spécifique pour le créer. Par exemple, en C++ la fonction `async` est utilisée pour obtenir un *future*. De façon symétrique, un *future* est implicite par création s'il n'existe pas d'instruction spécifique pour le créer. Ainsi, dans la catégorie de langages que l'on nomme « langages à objets actifs », appeler une méthode sur un objet actif produit un *future* qui contiendra, à terme, la valeur renvoyée par la méthode. [19]

Accès Un *future* est explicite d'accès s'il existe un type dédié pour le représenter. Par exemple, en Encore on trouve le type `Fut[T]` qui représente un *future* sur `T`. Par conséquent, il n'est pas possible de manipuler un *future* sur `T` de la même manière qu'une instance de `T`. Un *future* sur `T` est implicite d'accès s'il est possible de le manipuler de la même façon qu'une instance de `T`. Par exemple, en MultiLisp il est possible d'appliquer l'opérateur + sur un *future* sur `int`.

Typage Un *future* est explicite par typage s'il est possible de déterminer, à partir de son type statique, si une variable est de type *future*. En Encore, un *future* est une variable de type `Fut[T]`. Symétriquement, un *future* est implicite par typage s'il n'est pas possible de déterminer, à partir de son type statique, si une variable est de type *future*. Par exemple, dans le *framework* ProActive (Java) une variable de type statique `Foo` peut être une véritable instance du type `Foo` ou un *future* sur `Foo`. [20] [21]

La table 2 présente un récapitulatif de la distinction explicite / implicite pour chaque aspect, accompagné d'exemples. `o` désigne un objet, `m` une méthode sur cet objet, `f` un *future* sur `int`, `f2` un *future* sur `Foo`.

	Implicite		Explicite	
Aspect	Langage	Exemple	Langage	Exemple
Création	ProActive	`o.m()`	C++	`async(o, m)`
Accès	MultiLisp	`(+ f 1)`	Encore	`get(f) + 1`
Typage	ProActive	`Foo f2`	Encore	`f2 : Fut[Foo]`

TABLE 2 – Illustration de la différence explicite / implicite sur les aspects création, accès et typage des *futures*

L'aspect implicite (resp. explicite) de l'accès détermine l'aspect implicite (resp. explicite) de la synchronisation. Comme nous l'avons vu, la fonction `get` déclenche une synchronisation explicitement. Dans le cas de l'accès implicite, la synchronisation se produit lorsque le *future* est utilisé. Dans l'exemple Multilisp donné dans la table 2, `(+ f 1)` déclenche une synchronisation car il est nécessaire d'évaluer la valeur de `f` pour effectuer l'opération [22]. En revanche, transmettre `f` à une autre fonction (`(fn f 1)` par exemple) ne déclencherait pas de synchronisation, car la valeur de `f` n'est pas nécessaire pour effectuer cette opération.

C Traduction en C d'un appel asynchrone Encore

Nous présentons ici la traduction en C d'un appel asynchrone Encore, en complément du graphe d'appel de fonctions présenté en section 2.3.4. Le code exemple est présenté dans le listing

19. La création implicite de *futures* ne se limite pas aux langages à objets actifs
20. En termes techniques, il serait plus correct de dire « la variable peut se comporter comme un *future* sur `Foo` » au lieu de « la variable est un *future* sur `Foo` »
21. Dans un langage à typage dynamique, tel que Python, les *futures* sont nécessairement implicites au niveau du typage
22. Par « évaluer la valeur de f » nous entendons « Connaître la valeur avec laquelle f est résolu »

10. Pour rappel : un objet actif de type `Foo` est créé, stocké dans la variable `o`. La méthode `m` de cet objet actif est appelée de façon asynchrone, avec en paramètre la valeur `12`. Le résultat de cet appel est stocké dans la variable `f`, qui est donc un *future*.

L'ensemble de fragments de code 21, 22 et 23 forme la traduction C. Sont omises les lignes permettant de gérer la mémoire. Les noms des fonctions et des variables ont été raccourcis par soucis de clarté.

Listing 21 – Traduction C d'un appel asynchrone Encore

```
1  future_t* _fut_8 = Foo_m_future(_ctx, _o_5, NULL, _literal_6);
```

Le listing 21 représente l'invocation de la version *future* de la méthode `m`, en C. `_o_5` est l'objet actif sur lequel la méthode est appelée, `_literal_6` est la variable contenant le `12`.

Listing 22 – Envoi C d'un message

```
1  future_t* Foo_m_future(
2    pony_ctx_t** _ctx,
3    Foo_t* _this,
4    pony_type_t** runtimeType,
5    int64_t _enc__arg_x)
6  {
7    future_t* _fut = future_mk(_ctx, ENCORE_PRIMITIVE);
8    fut_msg__Foo_m_t* msg = pony_alloc_msg(POOL_INDEX(sizeof(fut_msg__Foo_m_t)),
         FUT_MSG__Foo_m);
9    msg->f1 = _enc__arg_x;
10   msg->_fut = _fut;
11   pony_sendv((*_ctx), ((pony_actor_t*) _this), ((pony_msg_t*) msg));
12   return _fut;
13 }
```

Le listing 22 présente le code C de la version *future* de `m`. `future_t` représente le type *future* dans le *runtime*, `fut___Foo_m_t` est la structure représentant le message « Appel asynchrone à la méthode `m` des objets actifs de type `Foo` ». On peut observer que la variable `msg` est remplie avec le *future* créé et la variable `__enc__arg_x`, qui représente le paramètre de la méthode. `FUT_MSG__Foo_m` est l'identifiant du message. Enfin, la fonction `pony_sendv` permet d'envoyer un message.

Listing 23 – Traitement C d'un appel asynchrone

```
void dispatch_Foo(pony_ctx_t** _ctx,
  pony_actor_t* _a,
  pony_msg_t* _m)
{
  // ...
  switch (_m->id)
  {
    // ...
    case FUT_MSG__Foo_m:
    {
      future_t* _fut = _m->_fut;
      int64_t arg_x = _m->f1;

      pony_type_t* methodTypeVars[] = {};

      encore_arg_t res;
      res.p = Foo_m(_ctx, _this, methodTypeVars, arg_x);

      future_fulfil(_ctx, _fut, res);
      break;
    }
  }
}
```

Le listing 23 présente le cas correspondant au traitement du message « Appel asynchrone de la méthode `o` d'un objet actif de type `Foo` » dans la fonction de *dispatch* de la classe `Foo`. La première partie correspond à l'extraction du *future* et des arguments. La seconde partie consiste en la résolution du *future*, *via* la fonction `future_fulfil`. Le *future* est résolu avec la valeur renvoyée par l'appel à la fonction `Foo_m`, qui est la version principale de la méthode `m` en C.

D Factorielle en Encore avec `async*`

Le listing 24 est une version purement fonctionnelle du listing 12. Les classes sont éliminées, et les appels à l'opérateur `!!` sont remplacés par des appels à l'opérateur `async*`.

Listing 24 – Factorielle asynchrone en Encore avec `async*`

```
import Task

fun fact(n : int) : Flow[int]
    return async*(fact_acc(1, n))
end

fun fact_acc(r : int, n : int) : Flow[int]
    if (n <= 0) then
        return r
    else
        return async*(fact_acc(r * n, n - 1))
    end
end

active class Main
    def main() : unit
        println(get*(async*(fact(10)))) -- 3628800
    end
end
```

E Fonction Haskell `checkOneBindingForFlow`

Termes utilisés Étant donné que cette fonction Haskell raisonne sur les types Haskell utilisés pour annoter l'AST, nous utiliserons les types Haskell dans notre explication. Nous rappelons que le type Encore `Flow` est désigné par le type Haskell `FlowType`. Le type « variable de type » (ou « paramètre générique ») est désigné par le type Haskell `TypeVar`.

Nous présentons la fonction dans le listing 25.

La fonction `checkOneBindingForFlow` prend en paramètre une paire de types, `param` et `arg`. La fonction renvoie `true` si `param` désigne le type `TypeVar` et si `arg` désigne soit le type `FlowType`, soit le type `TypeVar` que l'on peut inférer comme désignant le type `FlowType`.

Nous retrouvons les trois cas que nous avons identifiés précédemment en section 3.4.2 (augmentés à quatre afin de traiter les classes génériques). Chaque cas est accompagné d'un exemple. Dans chaque exemple, la fonction appelante est `g`, la fonction appelée est `f`. L'algorithme entier serait appliqué à chaque appel à `f`. Chaque exemple illustre le passage à `Flow` du paramètre générique de `f` de façon implicite (par propagation du paramètre de `g`) et de façon explicite (par utilisation d'une valeur de type statique `Flow`). La fonction Encore `id` désigne la fonction identité.

- Dans le cas où `param` est de type `TypeVar`, la fonction renvoie `true` si `arg` est de type `FlowType`, ou si la variable `flowCtx` est à `true`. Si `flowCtx` est à `true`, cela veut dire que nous sommes en train de traduire la version `Flow` d'une méthode générique, dont l'unique paramètre générique a pris le type Encore `Flow`. En conséquence, le paramètre générique de la méthode / fonction traduite et le paramètre générique de la méthode /fonction appelée sont identiques. Le listing 26 présente ces cas de figure ;
- Dans le cas où `param` désigne un type paramétré *builtin* (`hasResultType`), nous déclenchons une récursion. Considérons que le type `param` est de la forme `T[R]`, avec `T` le type générique lui-même et `R` le type de son paramètre. Considérons que le type `arg` est de la forme `T'[R']` [23]. Nous effectuons une récursion pour vérifier si un `TypeVar` apparaît dans `R` et si une valeur de type `FlowType` est passée à ce `TypeVar` (ou si `flowCtx` est à `true`). Le listing 27 présence ce cas de figure ;
- Dans le cas où `param` désigne une classe générique, nous procédons comme pour les types paramétrés *builtins*, à ceci près que nous effectuons la vérification sur tous les types paramétrant la classe. Le listing 28 présente ce cas de figure ;
- Dans le cas où `param` désigne une continuation, nous effectuons également une récursion. Considérons que la continuation `param` est de la forme `a -> b -> c`, `a` et `b` désignent les types des paramètres, `c` le type du résultat. Considérons que la continuation `arg` est de la forme `a' -> b' -> c'`. Nous effectuons une récursion sur toutes les paires (`a`, `a'`), (`b`, `b'`) et (`c`, `c'`) pour vérifier si on y trouve un `TypeVar` et s'il devient `Flow`. Le listing 29 présente ce cas de figure.

Dans chacun de ces exemples nous utilisons l'expression « *bind* » pour référer au fait que le type générique `t` prend une valeur-type. On dit que `t` est *bindé* sur ce type. Par exemple, invoquer la fonction identité (fonction générique sur le type `t`) avec un `int` en paramètre effectif *bind* le paramètre générique `t` sur le type `int`.

23. Le *typechecking* nous assure que `T'` est soit `T`, soit un sous-type de `T`. Il en va de même pour `R'` vis-à-vis de `R`

Listing 25 – Fonction `checkOneBindingForFlow`

```
checkOneBindingForFlow :: (Ty.Type, Ty.Type) -> Bool
checkOneBindingForFlow (param, arg)
  | Ty.isTypeVar param     = Ty.isFlowType arg || flowCtx
  | Ty.isClassType param   =
      any checkOneBindingForFlow $
            zip (Ty.getClassParameters param)
                (Ty.getClassParameters arg)
  | Ty.isArrowType param   =
      checkOneBindingForFlow (Ty.getResultType param,
                              Ty.getResultType arg) ||
      (any checkOneBindingForFlow $
             zip (Ty.getArrowParameters param)
                 (Ty.getArrowParameters arg))
  | Ty.hasResultType param =
      checkOneBindingForFlow (Ty.getResultType param,
                              Ty.getResultType arg)
  | otherwise              = False
```

Listing 26 – Exemple de *bind* de type générique pour un paramètre de type statique `t`

```
fun f[t](x : t) : int
    42
end

fun g[t](x : t) : int
    f(x)
    -- Asynchronous flow call to id : Flow[int]
    f(async*(id(3)))
end
```

Listing 27 – Exemple de *bind* de type générique pour un paramètre de type *builtin*

```
fun f[t](x : Fut[t]) : int
    42
end

fun g[t](x : t) : int
    var future = async(id(x))
    f(future)
    -- Asynchronous flow call to id. The result
    -- is wrapped in a future through an async
    -- call to id : Fut[Flow[int]]
    f(async(id(async*(id(2)))))
end
```

Listing 28 – Exemple de *bind* de type générique pour un paramètre de type classe paramétrée

```
1  active class Foo[t,u]
2      var _x: t
3      var _y: u
4
5      def init(x : t, y : u) : unit
6          this._x = x
7          this._y = y
8      end
9  end
10
11 fun f[t](x : Foo[t,int]) : int
12     42
13 end
14
15 fun g[t](x : t) : int
16     var foo = new Foo[t,int](x, 42)
17     f(foo)
18
19     var flow = async*(id(42))
20     var bar = new Foo[Flow[int],int](flow, 42)
21     f(bar)
22 end
```

Listing 29 – Exemple de *bind* de type générique pour un paramètre de type continuation

```
1  fun f[t](x : int -> t -> int) : int
2      42
3  end
4
5  fun cont[t](x : int, y : t) : int
6     x
7  end
8
9  fun g[t](x : t) : int
10     f(cont, x)
11     f(cont, async*(id(42)))
12 end
```