Praetonus

Zeste de savoir

Découverte du langage Pony

12 août 2019

Table des matières

Bonjour et bienvenue. Dans cet article, nous allons parler du langage Pony , un jeune langage fort sympathique.

Pony est un langage open-source focalisé principalement sur la programmation concurrente sûre et rapide. Dans cet article, nous explorerons les moyens de Pony pour accomplir cet objectif, en plus de détailler les bases de la syntaxe et de la sémantique du langage. Nous verrons aussi les avantages et inconvénients de Pony par rapport à des langages similaires.

Afin de pouvoir suivre cet article sans trop de difficultés, il est recommandé au lecteur de posséder une certaine assise dans au moins un langage de programmation. Être familier avec les principes et problèmes de la programmation concurrente peut également être bénéfique, même si nous détaillerons quelques points fondamentaux.

1. Préambule, la programmation concurrente

Avant d'entrer dans le vif du sujet et de présenter le langage Pony, faisons un détour pour parler de la programmation concurrente, le principal domaine visé par Pony.

La programmation concurrente consiste en la composition de différentes tâches s'exécutant indépendamment dans un même système, tout en pouvant communiquer et se synchroniser entre elles. L'implémentation réelle derrière cette notion de tâche peut varier (*thread*, acteur, coroutine, etc.), pour simplifier nous considérerons les tâches comme des fonctions et les systèmes comme des programmes dans les exemples de cette section. La programmation concurrente a plusieurs avantages, notamment le fait de pouvoir paralléliser le système en exécutant plusieurs tâches en même temps (par exemple, exécuter une tâche sur chaque cœur d'un processeur).

Un point important de la programmation concurrente est l'indépendance des tâches et de leur exécution, comme illustré ci-dessous.

```
fonction tache_A
  action_A1
  action_A2
  action_A3

fonction tache_B
  action_B1
  action_B2
  action_B3
```

Nous avons ici deux tâches, que nous allons exécuter de manière concurrente sans synchronisation. Le déroulement de l'exécution n'est pas déterministe, on peut donc obtenir un déroulement différent à chaque lancement du programme. Par exemple, quelques cas possibles :

```
cas 1          | cas 2          | cas 3
  action_A1    |     action_B1  |   action_A1
  action_A2    |   action_A1    |   action_A2
    action_B1  |     action_B2  |   action_A3
  action_A3    |   action_A2    |     action_B1
    action_B2  |     action_B3  |     action_B2
    action_B3  |   action_A3    |     action_B3
```

De plus, l'exécution des différentes sous-tâches ne forme pas forcément un entrelacement, il est possible que plusieurs sous-tâches (appartenant à différentes tâches) s'exécutent en même temps.

Bien que cette indépendance offre une grande flexibilité, elle est aussi la source de la majorité des problèmes introduits par la programmation concurrente. Par exemple, que se passerait-il si, étant donné une donnée accessible par plusieurs tâches, une des tâches accédait à cette donnée alors qu'une autre tâche est en train de la modifier ? En général, on ne peut pas prévoir. Ce

genre de chose s'appelle une *data race* (ou *race condition*) et est la source de l'écrasante majorité des bugs dans les programmes concurrents.

Pour éviter ces problèmes, il existe plusieurs méthodes.

— Synchroniser les accès aux données partagées. Il s'agit ici d'utiliser différents procédés (*mutex*, opérations atomiques, etc.) permettant d'éviter la modification d'une donnée en même temps que son accès par une autre tâche. Cette méthode est la plus courante dans les langages comme C et introduit une ribambelle de nouveaux problèmes complexes que nous ne détaillerons pas ici.
— Interdire totalement les données mutables. Puisqu'une *data race* peut se produire uniquement lors de la modification d'une donnée, ne pas disposer du concept de modification règle le problème. C'est la méthode de choix dans les langages fonctionnels.
— Interdire les données mutables partagées. Légère variante du point précédent, on est ici plus permissif sur les données locales à une tâche. Il s'agit de l'approche choisie par Pony.

2. Pony en quelques mots

Si l'on devait résumer Pony en quelques caractéristiques, le langage serait à acteurs, orienté objet, *capability secure*, statiquement et fortement typé, avec quelques éléments fonctionnels. Si ces termes ne vous sont pas familiers, pas d'inquiétude, nous détaillerons par la suite. L'implémentation de référence du langage est compilée et fonctionne actuellement sur x86 et ARM (32 et 64 bits dans les deux cas).

Le langage est le fruit du projet de doctorat de Sylvan Clebsch. Il a commencé à travailler sur les concepts derrière Pony en 2011 et le langage est open-source depuis 2015. C'est donc un langage très jeune, mais issu de récents travaux de recherche académique. Les plus grandes inspirations de Pony ont été les travaux sur les acteurs (par Carl Hewitt à l'origine[1] et étendus par Gul Agha[2]) et sur la sécurité basée sur les *capabilities* (formalisée par Mark Miller[3]). La thèse de S. Clebsch, bien qu'actuellement non finalisée, est disponible sur Github .

La philosophie Pony est « *get stuff done* », que l'on pourrait approximer par « faire et achever des choses ». Le langage est un outil pour résoudre des problèmes spécifiques. En particulier, l'un des problèmes visés par Pony est « comment construire un modèle de concurrence sûr et performant ? » Les armes de Pony pour répondre à ce problème sont séparées principalement en deux champs.

2.1. Un système de types puissant

Pony dispose d'un système de types offrant un certain nombre de garanties. En particulier :

— Le typage est statique. Autrement dit, le compilateur connaît le type de chaque objet à chaque instant du programme. Cela permet des vérifications de types très poussées à la compilation ;
— Le typage est fort. La conversion entre types se fait toujours de manière explicite et les opérations classiques (arithmétique, concaténation de chaînes de caractère, etc) se font généralement sur des objets de même type, afin de réduire les erreurs d'inattention. De plus, il est impossible de contourner les garanties proposées. En particulier, le forgeage

de référence est interdit. Si un objet est accessible dans une fonction, il y a été créé ou a été passé en paramètre. Ce dernier point participe à l'aspect *capability secure* du langage, terme sur lequel nous reviendrons ;
— Pas de *data races*, avec une vérification complète à la compilation via une partie du système de types nommée *reference capabilities*. Il s'agit également d'une propriété associée à la *capability security* de Pony ;
— Pas de pointeurs invalides, de dépassements de tampon, etc. Le pointeur nul n'existe pas et un programme ne plante jamais, sauf dans les cas extrêmes d'épuisement de la mémoire disponible ;
— Toutes les exceptions doivent obligatoirement être traitées.

Ce système de types est également très expressif, avec notamment la présence de sous-typage structural, de types algébriques et de types génériques. Nous détaillerons tout cela par la suite.

2.2. Un modèle naturellement concurrent

Pony est un langage à acteurs (concrétisation des « tâches » de la section précédente). Un acteur est assez similaire à une classe (au sens programmation orientée objet) mais dispose de propriétés supplémentaires très intéressantes. Voyons cela à travers un *Hello world* en Pony.

```

```

À la première ligne, nous déclarons un acteur nommé `Main` via le mot-clé `actor`. Vient ensuite la déclaration du constructeur `create` avec le mot-clé `new`. Un programme Pony commence dans le constructeur de l'acteur `Main`. Ce constructeur prend un paramètre de type `Env`, qui contient entre autres les arguments de la ligne de commande et les accès à l'entrée et à la sortie standard (les variables globales n'existent pas en Pony, on peut donc obtenir ces éléments uniquement via un paramètre). Le corps du constructeur, comme vous l'aurez sûrement deviné, permet d'afficher « Hello world ! » sur la sortie standard. Cette ligne contient néanmoins une particularité.

? Quel est le type de `env.out` et comment est déclaré `env.out.print` ?

Vous aurez probablement du mal à répondre à cette question, voici donc la réponse. `env.out` est du type `StdStream` et `StdStream.print` est déclaré de la manière suivante.

```

```

Nous avons donc affaire à un autre acteur et à un nouveau mot-clé, `be`. Celui-ci permet d'introduire un *behaviour*, qui est une fonction asynchrone. Lorsqu'un *behaviour* est appelé, celui-ci n'est pas exécuté immédiatement. À la place, un message est envoyé à l'acteur correspondant, qui exécutera son *behaviour* après avoir reçu ledit message. Pour en revenir à notre histoire de *Hello world*, l'appel de `print` envoie un message à l'acteur `env.out`, qui exécutera le *behaviour* après avoir reçu ledit message. Le texte n'est donc pas affiché immédiatement, mais peut-être

longtemps après que `Main.create` ait fini son exécution. Un constructeur d'acteur est un *behaviour*, la création d'acteurs est donc également asynchrone.

Quelques propriétés sont à noter à propos des acteurs et de leurs *behaviours*.

— Un acteur donné ne peut pas exécuter plus d'un *behaviour* au même moment. Autrement dit, le traitement des messages reçus est séquentiel ;
— Un nombre indéterminé de *behaviours* (sur des acteurs distincts) peuvent s'exécuter simultanément ;
— Un acteur donné ne peut accéder qu'à son propre état, jamais à l'état d'autres acteurs ;
— Pendant son exécution, un *behaviour* ne peut jamais observer de modification qu'il ne réalise pas lui-même.

L'acteur est donc l'unité de la concurrence, séquentiel seul, massivement concurrent à plusieurs. Ce paradigme permet d'exprimer naturellement la concurrence sans s'encombrer de la gestion manuelle de *threads*.

Une autre propriété des acteurs de Pony est la causalité des messages. Derrière ce nom compliqué se cache un principe simple. Si l'on dispose de trois acteurs, `A`, `B` et `C`, que `A` envoie un message à `B` puis un message à `C`, et qu'en réponse à ce message `C` envoie un message à `B`, alors `B` recevra le message de `A` avant le message de `C`. Plus généralement, il s'agit d'une relation de cause à effet où, pour chaque acteur, chaque message envoyé ou reçu est causé par tous les messages précédemment envoyés ou reçus, si les origines et destinations correspondent.

Les messages asynchrones sont le seul moyen de communication entre acteurs. Il n'y a pas d'état partagé ou *futures* bloquants. Le système ne peut donc jamais se retrouver en situation d'interblocage.

Il n'existe pas de terme courant pour la distinction modèle/instance (à la manière de classe/objet) pour les acteurs, et « acteur » peut désigner un modèle ou une instance. Vous verrez parfois le terme « objet actif » pour une instance d'acteur et « objet passif » pour une instance de classe.

3. Et pour quelques types de plus

Le système de types de Pony est très riche. Cette section présente les principales catégories de types et des exemples de cas d'utilisation.

3.1. Classes

En plus des acteurs, Pony dispose de classes, celles-ci étant introduites sans surprise par le mot-clé `class`.

1. C. Hewitt, P. Bishop, R. Steiger, *A Universal Modular Actor Formalism for Artificial Intelligence* (1973)
2. G. Agha, *Actors : A Model of Concurrent Computation in Distributed Systems* (1986)
3. M. Miller, K.-P. Yee, J. Shapiro, *Capability Myths Demolished* (2003)

La seule différence entre acteurs et classes et que ces dernières ne peuvent pas disposer de *behaviour*, mais uniquement de fonctions synchrones. On utilisera une classe plutôt qu'un acteur pour représenter une donnée « passive », qui ne peut pas recevoir de messages.

Dans cette classe `Counter`, nous commençons par déclarer deux champs, `_count` et `step`. Si un champ débute par un tiret-bas, il est privé, dans le cas contraire il est publique. Un champ (ou une variable locale de fonction) déclaré `var` est ré-assignable tandis qu'un champ déclaré `let` ne l'est pas.

Un constructeur, introduit par `new`, nommé `create` ici, initialise simplement les champs. Tous les champs d'un objet doivent être initialisés dans le constructeur, ou le programme ne compilera pas. Une classe ou un acteur peut avoir plusieurs constructeurs, dont les noms sont libres.

Une fonction introduite par `fun`, comme `value`, est une fonction classique, synchrone. Les acteurs peuvent aussi disposer de fonctions. Détail syntaxique, la valeur de retour d'une fonction est la dernière expression de son corps.

La déclaration de la fonction `increment` contient un `ref`. Pour simplifier, cela signifie que la fonction souhaite modifier son receveur, et que celui-ci doit donc être mutable dans la fonction appelante (par défaut, une fonction ne peut pas modifier son receveur). En réalité, `ref` est une *reference capability* et a des implications complexes sur lesquelles nous reviendrons.

Il n'est pas possible d'hériter d'une autre classe, Pony préférant la composition à l'héritage. Pour compenser, il existe un système d'interfaces et de traits, dont nous allons parler dans quelques instants.

3.2. Primitives

Il existe un troisième membre dans le club des types de base de Pony, la primitive, introduite par le mot-clé `primitive`. Une primitive peut disposer de fonctions, mais pas de champs. De plus, il n'existe qu'une instance de chaque primitive. Cela signifie que le constructeur d'une primitive donnée renverra toujours la même référence. Les primitives ont plusieurs utilités.

— Une valeur singleton. Par exemple, la bibliothèque standard contient la primitive `None`, utilisée pour les objets « sans valeur » ;
— Une énumération, en formant une union de plusieurs primitives. Nous présenterons les unions par la suite ;
— Une collection de fonctions. Les fonctions globales n'existent pas en Pony. Une primitive peut être utilisée pour regrouper des fonctions sur le même thème.

3.3. Interfaces et traits

Les interfaces et les traits sont très similaires, ils permettent de définir une liste de fonctions dont un type doit disposer pour implémenter l'interface ou le trait.

```
```

Une interface introduit une relation de sous-typage structurel. Tout type, passé, présent ou futur, qui dispose d'une fonction `fun string(): String` est `Stringable`.

Le sous-typage introduit par un trait est nominal. Un type doit explicitement spécifier qu'il est un `Adder`, même si il dispose d'une fonction `fun ref add(n: U64)`.

```
```

Ici, `A` est un `Adder`, ce qui n'est pas le cas de `B`.

Le choix entre un trait et une interface dépend de ce que l'on souhaite faire du type. Un trait empêche l'inclusion accidentelle de types et permet donc de contrôler toutes les relations de sous-typage si l'on a besoin de cette sûreté. À l'inverse, une interface est beaucoup plus flexible et peut par exemple être utilisée avec un type défini dans une bibliothèque externe.

3.4. Types composés

Pony dispose de plusieurs sortes de types composés (ou algébriques).

- Les n-uplets. Un n-uplet est une séquence de types. Par exemple, `("ABC", U64(42))` est un objet de type `(String, U64)`. On peut utiliser un n-uplet pour retourner plusieurs valeurs d'une fonction, par exemple.
- Les unions. Un objet du type union `(A | B)` est soit de type `A`, soit de type `B`. Un exemple d'application de ce genre de type est un type optionnel, par exemple `(MonType | None)`.
- Les intersections. Un objet du type intersection `(A & B)` est à la fois du type `A` et du type `B`. Ce genre de types est principalement utilisé en tant que contrainte sur les types génériques.

3.5. Types génériques

Pony permet de définir des types génériques, soit des types paramétrés selon d'autres types. La généricité peut se trouver au niveau d'un type ou au niveau d'une fonction. Par exemple, le type `Array` de la bibliothèque standard est défini comme ceci.

```
```

Ici, `A` est le type générique. On peut également poser des contraintes sur un type générique.

```
```

`A` doit donc obligatoirement être un sous-type de `Stringable`. Les opérations possibles sur un type générique correspondent aux opérations possibles sur sa contrainte, et un type sans contrainte dispose de très peu d'opérations possibles.

4. Où l'on parle d'expressions

4.1. Opérations de base

Pony dispose des opérations arithmétiques et logiques classiques. Une différence importante est néanmoins à noter par rapport à la plupart des langages : la priorité des opérateurs n'existe pas et il faut placer des parenthèses dans toutes les expressions à plus de deux opérateurs. La raison derrière cela est la volonté de réduire la charge cognitive sur le programmeur, les règles de priorité des opérateurs étant assez compliquées à retenir parfaitement (que la personne n'ayant jamais fait de `man operator` jette la première pierre).

Parlons également des comparaisons. Il est possible de réaliser un test d'égalité structurale (opérateur `==`, ces objets contiennent-ils les mêmes données ?) ou identitaire (opérateur `is`, ces références désignent-elles le même objet ?).

4.2. Structures de contrôle

Comme tout langage impératif qui se respecte, Pony dispose de conditions.

```
```

Un bloc `if` est une expression et renvoie la valeur de la dernière expression de la branche empruntée. Si il n'y a pas de `else` et qu'aucune branche n'est prise, `None` est renvoyé.

Concernant les boucles, on en trouve plusieurs sortes.

```
```

On peut utiliser `break` et `continue` pour respectivement sortir d'une boucle et passer à l'itération suivante. Toutes ces boucles sont des expressions et peuvent disposer d'un bloc `else` renvoyant une valeur si on n'entre pas dans la boucle, ou si on sort de la boucle avec un `break` sans opérande.

4.3. Chaînage de méthodes

Le chaînage de méthodes permet de réaliser plusieurs appels successifs sur un objet sans que la méthode ait à renvoyer son receveur.

```
```

4.4. Filtrage de motifs

Pony dispose d'un mécanisme de filtrage de motifs, où il est possible de filtrer sur des valeurs et sur des types, avec des gardes et des captures.

```
```

Une expression `match` renvoie la dernière expression de la branche empruntée.

4.5. Objets littéraux

Parfois, créer un objet d'un type anonyme à la volée s'avère très pratique. Pony a le bon goût de proposer cela.

```
```

Créer des lambdas est également possible (il s'agit en fait d'un sucre syntaxique pour un objet littéral).

```
```

Cette lambda prend une `String` en paramètre et capture `env` depuis le champ lexical de la fonction où l'on se trouve.

4.6. Application partielle

Toujours dans les éléments fonctionnels, on peut trouver l'application partielle de fonctions.

```
```

Évaluer `part(5)` revient à évaluer `obj.foo(2, 5)`.

4.7. Exceptions

Les exceptions en Pony sont particulières, elles n'ont ni type ni valeur. Elles sont levées par la directive `error` et rattrapées dans un bloc `try .. else .. end`.

```
```

Il n'est pas obligatoire de traiter l'exception dans la fonction où elle est levée. Pour cela, la fonction doit être marquée partielle avec le signe `?`. L'appel d'une fonction partielle doit se

trouver dans un bloc `try` ou dans une autre fonction partielle. Un *behaviour* ne peut pas être partiel, ce qui fait qu'une exception ne peut pas s'échapper et causer une erreur d'exécution.

```
```

5. Reference capabilities, vous dites ?

Avant de parler *reference capabilities*, parlons simplement *capabilities*.

Dans la sécurité basée sur les *capabilities*, une *capability* est un élément non-forgeable qui associe une référence vers une entité à l'ensemble des opérations autorisées sur l'entité à travers la référence. Cela permet de limiter l'autorité ambiante d'une référence (les actions possibles avec une autorisation implicite) et d'expliciter les droits dont dispose cette référence. Autrement dit, pour pouvoir effectuer une action, un élément d'un système doit en avoir obtenu l'autorisation d'un autre élément ayant lui même le droit d'effectuer l'action. Ce concept a été utilisé au départ dans les systèmes d'exploitation ; par exemple un descripteur de fichier est une référence associant un fichier à un ensemble de droits d'accès, droits cédés ou non en fonction de l'identité de l'utilisateur.

On retrouve ce concept, souvent inconsciemment, dans la plupart des langages de programmation. En effet, un objet typé est une *capability* qui associe un emplacement en mémoire à un ensemble de fonctions appelables sur l'objet. On parle d'*object capability*, concept que le langage E a été le premier à définir formellement. Un langage (ou un système en général) est dit *capability secure* si les règles des *capabilities* sont incontournables. Par exemple, le C n'est pas *capability secure* car on peut forger des références via le transtypage de pointeurs.
Pony fait partie des langages *capability secure*, à la fois dans ses *object capabilities* et dans ses *reference capabilities*, dont nous allons parler immédiatement.

La littérature francophone sur les *capabilities* étant très réduite, il n'existe pas de traduction standard pour le vocabulaire du domaine.

5.1. Les *reference capabilities* à proprement parler

Les *reference capabilities* de Pony sont des *capabilities* qui permettent de prouver l'absence de *data races* à la compilation. Ce sont des annotations de type (à la manière de `const` en C++) ; et elles sont au nombre de six : `iso`, `trn`, `ref`, `val`, `box` et `tag`.

— Une référence `iso`, `trn` ou `ref` est mutable, elle permet de lire, écrire et de connaître l'identité de l'objet associé (opérateur `is`) ;
— Une référence `val` ou `box` est immutable, elle permet de lire et de connaître l'identité de l'objet associé ;
— Une référence `tag` est opaque, elle permet uniquement de connaître l'identité de l'objet associé.

Ces propriétés sont déjà intéressantes, mais la force des *reference capabilities* vient de ce qu'elles interdisent plutôt que de ce qu'elles autorisent aux alias d'une référence. Deux références sont des alias si elles ont la même identité (si elles désignent le même objet). Du point de vue d'une référence donnée, un alias est local si il se trouve dans le même acteur, et distant si il se trouve dans un autre acteur. Pour chaque *reference capability*, on introduit des règles sur les types d'alias qui ne peuvent pas exister. En terme de *capabilities*, une *reference capability* associe une référence vers un objet à l'ensemble des alias interdits pour cette référence. Pour nos *reference capabilities* et avec une référence `R` se trouvant dans un acteur quelconque :

- Si `R` est une référence `iso`, aucun alias en lecture ou en écriture ne peut exister, local ou distant. Un objet référencé par une référence `iso` est profondément isolé, aucun objet en dehors de la « bulle » composée de l'`iso` et de ses sous-objets ne peut accéder à quelque chose dans la « bulle », et vice-versa ;
- Si `R` est une référence `trn`, les alias distants en lecture et en écriture ainsi que les alias locaux en écriture ne peuvent pas exister. On peut utiliser `trn` pour construire une structure de données au fur et à mesure, puis « abaisser » la référence en une référence immutable. Cela permet notamment la création de structures de données cycliques immutables ;
- Si `R` est une référence `ref`, les alias distants en lecture et en écriture ne peuvent pas exister. `ref` est le plus proche d'une référence dans un langage orienté objet classique comme Java et peut être aliasé très librement dans l'acteur courant ;
- Si `R` est une référence `val`, les alias locaux ou distants en écriture ne peuvent pas exister. Là où `iso` est profondément isolé, `val` est profondément immutable : l'objet et ses sous-objets sont constants. De plus, cette immutabilité est irréversible, un objet `val` (ou un de ses sous-objets) ne deviendra jamais mutable.
- Si `R` est une référence `box`, les alias distants en écriture ne peuvent pas exister. `box` représente l'immutabilité locale. La mutabilité réelle de l'objet importe peu, une référence `box` a uniquement besoin de lire. C'est un peu comme `const` en C++.
- Si `R` est une référence `tag`, aucun type d'alias n'est interdit. `tag` transporte simplement l'information sur l'identité d'un objet. Toutes les *reference capabilities* peuvent avoir un alias `tag`.

Et en image, la matrice d'interdictions :

		alias globaux		
		lect. et ecr. interdites	ecr. interdite	rien d'interdit
alias locaux	lect. et ecr. interdites	**iso**	S/O	S/O
	ecr. interdite	**trn**	**val**	S/O
	rien d'interdit	**ref**	**box**	**tag**
		référence mutable	référence immutable	référence opaque

Si tout cela n'est pas clair, le tableau suivant résume les différentes propriétés en terme d'alias autorisés plutôt qu'interdits.

référence	**alias local**		**alias global**		**alias t**
	lecture	écriture	lecture	écriture	
iso					
trn					
ref					
val					
box		ou			
tag					

À travers cela, deux propriétés générales se dessinent.

— Si un acteur dispose d'une référence en lecture sur un objet, aucun autre acteur ne peut disposer d'une référence en écriture ;
— Si un acteur dispose d'une référence en écriture sur un objet, aucun autre acteur ne peut disposer d'une référence en lecture.

Et voilà, ces propriétés, combinées au fonctionnement des acteurs, sont suffisantes pour s'assurer l'absence de *data races* grâce au système de types.

Afin de maintenir les garanties des *reference capabilities*, le langage impose des restrictions sur les types de références transmissibles entre acteurs (c'est à dire passables en paramètre de *behaviour*).

— `tag` n'offre aucun accès en lecture ou en écriture. La transmission de `tag` est donc possible ;
— `val` offre un accès en lecture et garantit qu'aucun accès en écriture n'existe dans le programme. La transmission de `val` est possible ;
— `iso` offre un accès en lecture et en écriture et garantit qu'aucun autre accès n'existe dans le programme. À condition que l'émetteur ne conserve pas de référence, la transmission d'`iso` est possible. Détruire une référence est possible grâce à un élément du langage nommé lecture destructive.

Plus généralement, il s'agit des *reference capabilities* qui interdisent la même chose aux alias locaux et aux alias globaux (la diagonale dans la matrice d'interdictions). En effet, dans ces cas là, l'acteur qui possède la référence n'entre pas en ligne de compte.

Il est donc possible de transmettre des données mutables, immutables ou opaques et ce sans aucune copie, ce qui est très important pour les performances. Les *reference capabilities* n'ont aucun surcoût à l'exécution, toutes les vérifications nécessaires étant réalisées à la compilation.

Pour faire un détour par les fonctions, le `ref` de `fun ref` est la *reference capability* `ref`. Cette annotation sur une fonction signifie donc en réalité que l'objet receveur de la fonction (`this`) doit être d'un type compatible avec `ref` lors de l'appel.

Concernant les acteurs, puisqu'un acteur ne peut pas examiner l'état des autres acteurs mais doit pouvoir examiner son propre état, un acteur voit tous les autres acteurs en tant que `tag` et se voit lui-même en tant que `ref` (`this` est `ref` dans un *behaviour*). Contrairement à d'autres

langages à acteurs, les acteurs sont donc entièrement intégrés dans le système de types en Pony.

Si vous êtes intéressés par les aspects formels des *reference capabilities*, vous pouvez vous diriger vers le papier à ce sujet.

6. Quand sûreté rime avec performances

Toutes ces garanties sont intéressantes à avoir, mais l'intérêt du langage serait fortement diminué si les performances s'en retrouvaient impactées. En réalité, la sûreté apportée permet l'implémentation très efficace de plusieurs éléments de l'environnement d'exécution.

6.1. En général

L'implémentation de référence de Pony est entièrement compilée. Les optimisations du compilateur (via LLVM) permettent la production de binaires très efficaces, notamment grâce à l'analyse d'alias très poussée fournie par les *reference capabilities*. En cas de besoin, il est possible d'appeler des fonctions C avec un système de FFI. Pony utilise l'ABI C pour ses propres fonctions, appeler une fonction C n'a donc aucun surcoût.

L'environnement d'exécution peut lancer un nombre arbitraire de planificateurs, chacun dans un *thread* système. Chaque planificateur possède d'une file d'acteurs disposant de *behaviours* à exécuter et traite cette file jusqu'à l'arrêt du programme. Un planificateur sans acteur à traiter peut en obtenir un depuis un autre planificateur. Tous ces algorithmes sont entièrement non-bloquants et sont synchronisés avec des opérations matérielles atomiques, ils sont donc très rapides. La répartition des acteurs sur différents planificateurs est ce qui permet la scalabilité virtuellement infinie de Pony.

Les objets transmis dans les messages ne sont jamais copiés, contrairement à d'autres langages à acteurs. Cela est possible grâce aux *reference capabilities*, qui garantissent l'inexistence d'un état mutable partagé.

6.2. Le ramasse-miettes

Le ramasse-miettes de Pony est un peu particulier. Il est entièrement concurrent, non-bloquant et basé sur les messages entre acteurs. Un acteur peut effectuer un cycle de collection pour libérer les objets inutilisés qu'il a alloué (même s'ils ont été transmis à d'autres acteurs) lorsqu'il n'est pas en train d'exécuter un *behaviour*. La collection ne requiert pas l'examen de l'état d'autres acteurs grâce aux garanties des *reference capabilities* et de la causalité des messages. Lors de l'exécution d'un *behaviour*, les opérations relatives au ramasse-miettes (comptage de références) sont réalisées uniquement lors de l'envoi de messages, ce qui fait que l'exécution d'un *behaviour* est toujours déterministe. Les messages du ramasse-miettes sont envoyés à la fin d'un cycle de collection et n'attendent pas de réponse. L'acteur est donc immédiatement disponible pour exécuter un nouveau *behaviour*, sans phase de synchronisation.

Une variante de l'algorithme est appliquée à la collection des acteurs eux-mêmes. Pour les connaisseurs d'Erlang ou autre, cela signifie qu'il n'y a pas besoin de *poison pills* en Pony ; un

acteur est supprimé lorsqu'il peut prouver qu'il n'exécutera jamais de nouveau *behaviour*. Ici aussi, tout est basé sur des échanges de messages. Un acteur spécial dédié à la détection de cycles entre acteurs est intégré au processus.

Les détails du ramasse-miettes sont disponibles dans plusieurs papiers, pour les objets et pour les acteurs .

Des benchmarks sur les performances de l'implémentation de Pony sont disponibles dans le papier sur les *reference capabilities*.

7. Pony est-il le bon outil pour vous ?

Bien que Pony se veuille généraliste, le langage est plus adapté à certaines catégories applications qu'à d'autres. Un domaine où Pony brille est une application hautement concurrente et asynchrone, avec des besoins de haute performance et de temps réel souple. Par exemple, une application de traitements financiers, un système de gestion de base de données ou un serveur de jeu vidéo multijoueur.

Les programmes Pony ne sont pas formellement vérifiés, mais les diverses garanties de sûreté offertes par le langage en font un bon candidat pour des systèmes critiques.

La bibliothèque d'exécution du langage (écrite en C) est utilisable indépendamment du compilateur pour les projets ne pouvant pas (ou ne souhaitant pas) utiliser le langage lui-même. Dans ce cas, les garanties des *reference capabilities* doivent être maintenues par le programmeur sans l'aide du compilateur.

Comparé à d'autres langages à acteurs, Pony a plusieurs avantages, que ce soit au niveau de la sûreté ou au niveau des performances. Par rapport à des langages comme Erlang, Pony ne copie jamais le contenu des messages et peut collecter les acteurs automatiquement. Par rapport à des langages comme Scala/Akka, Pony interdit le partage d'état mutable et rend donc les *data races* impossibles.
La garantie de causalité des messages peut être soit un avantage soit un inconvénient selon les besoins de votre application.

Pony est proche de Go dans la problématique visée, mais les deux langages divergent totalement dans leur réponse à cette problématique. Là où Pony est basé sur un système de communication entièrement asynchrone et non-bloquant, Go prend l'approche inverse avec des communications uniquement bloquantes. De plus, Go dispose de moins de garanties que Pony, les *data races* et l'interblocage étant possibles.

Certains domaines ne sont absolument pas adaptés à Pony.

— Les paradigmes synchrones et bloquants. À part via un appel FFI, réaliser une opération bloquante est impossible en Pony. Ce n'est de toute façon pas quelque chose de souhaitable étant donné que cela bloquera un des planificateurs et réduira donc le degré de parallélisme.
— La programmation système. Pony n'offrant aucun accès direct à la mémoire, une application système en Pony devra toujours être interfacée avec une couche écrite dans un langage disposant de ces accès.

— Le temps réel strict. En raison du fonctionnement des acteurs, il est impossible de garantir un délai entre l'envoi d'un message et le début de l'exécution du *behaviour* correspondant. En revanche, un *behaviour* isolé peut remplir une condition de temps réel strict, le ramasse-miettes n'interrompant jamais un *behaviour*.

Nous sommes arrivés au terme de cette présentation du langage Pony. N'hésitez pas à installer le compilateur , à jeter un œil au tutoriel officiel (bien plus détaillé que cette présentation rapide) et à essayer le langage. Vous pouvez aussi faire un tour sur la liste de diffusion ou le canal IRC .

Pony est un langage jeune et en pleine évolution. Le langage n'a pas encore atteint la version 1.0 et les API n'ont pas encore de garanties de stabilité. Néanmoins, il montre une certaine maturité en étant déjà utilisé en production dans des applications professionnelles (en finance, principalement).

De nombreuses évolutions sont à venir prochainement, en particulier la version distribuée de la bibliothèque d'exécution, qui permettra d'exécuter un programme Pony sur un cluster de manière transparente (le code source est indépendant du nombre de machines à l'exécution), et un système de types dépendants de valeurs, permettant d'utiliser des valeurs au lieu de types en paramètres génériques.

Certains concepts du langage, notamment les *reference capabilities*, sont compliqués à appréhender mais le raisonnement vient en général rapidement. D'ailleurs, si vous avez déjà fait une certaine quantité de programmation concurrente, vous pensez déjà *reference capabilities* sans le savoir !

Si vous souhaitez contribuer au langage, il y a toujours quelque chose à faire, que ce soit au niveau des nouvelles fonctionnalités (via un processus de RFC), des bugs à corriger ou de la documentation à améliorer.

Liste des abréviations

ABI Application Binary Interface. 13

FFI Foreign Function Interface. 13, 14

RFC Request For Comments. 15