Bonjour à tous, et merci d'avoir choisi cette session dans laquelle il ne sera question ni de Kubernetes,ni d'IA... Quoique ? Il y sera en revanche question fonctions anonymes et de leurs mise en œuvre. Et celà au travers de nombreux fragments de code qui devraient être suffisamment faciles à suivre. Dans tous les cas à la fin vous trouverez des pointeurs vers cette présentation elle même et le code associé sur GitHub.

Je suis Frédéric Cabestre, un artisan du logiciel indépendant issu il y a fort longtemps du monde académique... Ce qui a laissé quelques traces. Si vous avez des soucis avec des sujets ésotériques ou tout simplement besoinde faire accompagner une équipe de développement, vous pouvez me contacter... Vous trouverez bien comment
faire.

Les fonctions d'ordre supérieur (c'est-à-dire dont les paramètres sont des fonctions ou dont la valeur deretour peut être une fonction) font définitivement partie de l'outillage de base du développeur.

Pour celà les fonctions sont devenues des entités de première classe des langages de programmation. Autrement ditce sont des valeurs à part entière que l'on peut par exemple affecter à des variables.

Il n'est d'ailleurs même plus nécessaire de les nommer pour pouvoir les définir. Nous pouvons créer des fonctions anonymescommunément appelées « lambdas ».

Cet attirail permet d'exprimer plus l'intention que la procédure à suivre. Par exemple ici en disposant d'une fonction « filter » sur des collections, à laquelle on peut fournir un prédicat, il n'est plus besoin de s'appesantir sur les détails d'une itération. La démarche est plus déclarative que procédurale.

Dans ce qui suit l'essentiel des exemples de code seront en Kotlin. J'aurais pu m'appuyer sur C#, Rust, Java ou Javascript... Non, quand même pas Javascript ! Mais Kotlin c'est mon langage du moment, dont la lecture ne devrait pas vous troubler. Contrairement à...

... Common Lisp. Si Lisp va être une sorte de fil rouge historique de cette présentation, je n'avais pas envie de vous perdre bêtement en route. D'autant que, moi qui ai enseigné en TP de Lisp à la Fac il y a plus de 25 ans, j'en ai un peu bavé pour écrire cet exemple pourtant simple.

Dans cet exemple la partie en surbrillance est donc une lambda prenant en paramètre un entier et vérifiant s'il est pair (le reste de la division par 2 doit être 0).

Kotlin, comme beaucoup de langages aujourd'hui, dispose de capacités d'inférence de type. Ce qui permet dans une certaine mesure de s'économiser quelques annotations de type.

Particularité de Kotlin, on peut dans certains cas, comme ici, nommer implicitement un paramètre unique « it ».

Et autre particularité de Kotlin, une lambda en dernière position des paramètres d'une fonction peut être sortie de la liste des paramètres. C'est un emprunt à Groovy, je crois, qui contribue à faire de Kotlin un langage sympathique pour faire des DSL embarqués... Mais ça sera l'objet d'une autre présentation. Qui sait ?

En tout état de cause, cet exemple n'est pas des plus intéressants pour notre exploration...

Considérons plutôt celui-ci: un prédicat ajustable par son premier paramètre ( « x ») pour vérifier une propriété du second paramètre (« y »). Et tant qu'à

faire, cet ajustement est le résultat d'un calcul arbitrairement long effectué sur « x ».

Je vous l'accorde, c'est un exemple tiré par le peu de cheveux qu'il me reste. Mais il faut faire simple pour parler de choses compliquées.

Et si l'on veut utiliser ce prédicat à deux arguments, dans le même contexte que précédemment, il va falloir l'adapter, à l'aide d'une fonction anonyme...

... en fixant le premier argument pour le ramener à un seul paramètre à tester.

Un inconvénient de cette approche est que « longComputation » sera effectuée à chaque itération de « filter ».

Une façon de résoudre ce problème est de sortir ce long calcul de « parametricPredicate ». Mais j'ai envie de vous présenter une autre approche qui serait de pouvoir appliquer partiellement « parametricPredicate » à un argument. Et pour cela je dois vous parler de...

... Curryfication. Ce terme est dérivé du nom d'Haskell Curry, un logicien du 20e siècle ayant mené des travaux sur la logique combinatoire et qui donné son prénom à un célèbre langage fonctionnel. En réalité ce concept est plutôt dû à Moses Schönfinkel, mais Schönfinkelification aurait été plus difficile à prononcer.

Alors de quoi s'agit-il ?

Tout simplement de voir une fonction à 2 paramètres...

... comme équivalente à une fonction à un paramètre produisant un autre fonction à un paramètre...

... Et donc comme une fonction que l'on peut appliquer partiellement ou graduellement à plusieurs arguments. Évidemment on peut récursivement généraliser de principe à N paramètres.

Si l'on repart de notre example de travail, curryfier « parametricPredicate » revient à la transformer ainsi...

Que peut-on constater ici ?

Qu'elle ne prend plus qu'un seul paramètre, celui qui va servir à « longComputation ». Et que son type indique bien qu'elle retourne une fonction de Int vers Boolean.

Que l'on peut éventuellement affecter son résultat à une variable. N'oublions pas que les fonctions sont ici des citoyens de première classe...

Et donc qu'on peut utiliser ce résultat comme prédicat pour « filter » et accessoirement cette fois le long calcul n'est effectué qu'une seule fois et non à chaque itération du filtre.

Soit dit en passant, on aurait aussi pu faire un pas de plus et dire que « parametricPredicate » est...

... une variable qui reçoit une fonction de Int qui retourne une fonction de Int vers Boolean. Mais bon, restons là...

Avant d'aller plus loin, faisons un premier saut en 1958 à la naissance de LISP.

Cet élégant barbu est...

John McCarthy (1927-2011), chercheur de l'université de Standford ayant obtenu le prix Turing en 1971, il est co-inventeur des systèmes à temps partagé (pensez

Unix) et pionnier de l'IA (on lui doit le terme). En 1956, il crée l'algorithme d'élagage Alpha-Béta utilisé au cœur de moteurs de jeux pour les échecs par exemple.

Mais il a aussi contribué à la création de LISP en 1958 second langage concret après FORTRAN (1954).

Lisp est un langage conçu avant tout pour la manipulation symbolique, par opposition à FORTRAN dont le but et le calcul numérique. L'idée était d'avoir un langage facilitant l'écriture d'algorithmes tels que la dérivation symbolique de fonctions (une des premières applications) et largement adopté de ce fait dans la recherche en IA et en théorie des langages (de programmation ou naturels)...

Et de manière assez cocasse, c'était à l'origine un langage sans fonctions d'ordre supérieur avec une syntaxe proche de FORTRAN (le M-language) permettant de manipuler des structures de liste (le S-language). Rapidement, on s'est aperçu qu'il est possible de plonger le premier dans le second grace aux fonctions EVAL et APPLY, rendant LISP homo iconique. C'est-à-dire que la structure du langage et structure des données manipulées est identique. Ça ouvre une porte à la méta-programmation qui s'apparente à la réflexivité.

Dans la foulée, LISP gagne les fonctions d'ordre supérieur et la possibilité de définir des fonctions anonymes, des LAMBDAS. Un argument fonctionnel de fonction est ce que l'on nomme un FUNARG dans le jargon Lispien.

J'ai cru pendant longtemps de LISP était avant tout fonctionnel et avait été inspiré pas le lambda calcul. En fait LISP 1.5 était encore très impératif (SETQ, RPLACA...) et McCarthy aurait indiqué s'en être peu inspiré.

Sauf que dans les premières moutures de LISP, les FUNARG avaient un problème étudié (ici par Joel Moses dans un mémo de MIT) et longuement débattu.

Pour illustrer ce problème, reprenons notre exemple (pas en LISP bien sûr). C'est peu ou prou l'exemple initial simplifié, sans l'appel à « filter » et avec une variable « c » définie dans « main » pour les besoins de l'exemple.

En passant, je me permets une petite auto-promotion éhontée pour une de mes présentations passées qui parle des fonctions, éventuellement récursives, et de leur mise en œuvre. Présentation qui pourrait être un complément intéressant à celle-ci.

Nous allons suivre pas à pas le déroulement de ce programme. Comme je l'explique plus en détail dans mon autre présentation, l'environnement d'exécution d'une fonction est fourni par un enregistrement d'activation plus tous ceux des fonctions appelantes. Un enregistrement est créé à l'activation de la fonction et détruit quand elle se termine. Et dans la plupart des langages (mais pas tous) cette gestionse fait sous forme de pile.

Donc ici, on commence avec l'enregistrement d'activation de « main » dans la pile.

« c » se voit affecter la valeur 42

Puis on appelle la fonction « parametricPredicate » currifiée avec 3 en argument.

Ce qui génère un nouvel enregistrement d'activation pour elle sur la pile. À son tour « longComputation » est appelée avec 3 en argument.

Et un nouvel enregistrement d'activation lui est allouée. « longComputation » produit et retourne son résultat 9.

Qui est affecté à « c » dans l'environnement de « parametricPredicate ».

Et c'est là que les Athéniens s'atteignirent ! « parametricPredicate » doit retourner une fonction anonyme, et naïvement, on pourrait considérer (comme en langage C par exemple) qu'il s'agit simplement d'un pointeur sur fonction.

Ici l'adresse du code compilé de la fonction « @Lambda0 » est retourné et affecté à la variable « predicate » dans l'environnement de « main ».

Puis on appelle « predicate » avec 81 en paramètre à tester... Donc, on lui alloue un enregistrement d'activation.

Mais dans cet environnement « c » n'a pas de valeur, alors on remonte la chaîne des environnements pour en trouver une. On tombe sur le 42 initial. Et là, c'est le drame ! On s'attend à un résultat positif (81 est divisible par 9 en nombre entier), pourtant c'est faux parce que la lambda n'utilise pas la bonne valeur de « c ». Il pourrait même ne pas y avoir de valeur pour « c » !

À vrai dire, je ne sais pas ce qui est le pire entre un programme qui plante clairement ou qui répond à côté sans que l'on s'en aperçoive.

Alors ne nous méprenons pas : cela peut être considéré comme une fonctionnalité légitime. Ça s'appelle de la liaison dynamique. Et pendant longtemps, le sujet a été débattu, avec ses partisans et ces détracteurs. Et c'est même devenu un idîome LISP. On parle d'idîome dans un langage quand un design pattern est trèslié à la sémantique du langage.

Je serais enclin à dire, si l'on se place dans un cadre de programmation fonctionnelle, que le principe de raisonnement local est mis à mal par un tel mécanisme. Quand je code, j'ai du mal avec les effets contextuels d'utilisation d'une fonction autre que ses paramètres d'entrée.

Donc le sujet est débattu pendant plus de 15 ans dans le monde lispien. Jusqu'à l'arrivée de SCHEME.

Et à croiser la route de...

Guy Lewis Steel Junior, à qui on doit entre autres Emacs et le premier portage de TeX de Donald Knuth. Il a été impliqué dans la définition ou la normalisation de nombreux langages de programmation, comme *Lisp et C* chez Thinking Machines ; Common Lisp, C, Fortran dans le cadre de divers commités ; Et Java en tant qu'employé de Sun en 1994 puis Oracle depuis 2010. Il est même récemment allé travailler avec Epic Games sur le langage Verse en compagnie de Simon Peyton-Jones, un des piliers d'Haskell.

Pour ce qui nous concerne aujourd'hui, il est surtout le co-créateur de Scheme en 1975 de SCHEME avec Gerald Sussmann au MIT. Gerald Jay Sussmann auteur du livre « Structure and Interpretation of Computer Programs », que je vous invite à parcourir tant il fait partie des fondamentaux.

Il est aussi l'auteur des « λ the ultimate » papers. C'est une série d'articles parfois coécrits avec Sussmann, qui prend le partit de la liaison lexicale. Il s'agit de faire en sorte, même au prix d'un surcoût à l'exécution d'un programme, que les variables libres d'une lambda prennent la valeur qu'elles avaient au moment de leur définition et non au moment de l'exécution.

Et au travers de multiples exemples, il montre toute l'utilité de ce concept.

Pour la peine, la liaison lexicale est au fondement de l'interprétation du lambda calcul.

Et sa mise en œuvre concrète est depuis longtemps comprise, comme dans cet article fondateur de Peter Landin datant de 1964 ! Accessoirement lui doit aussi le terme « sucre syntaxique » !

Reprenons le déroulé de notre exemple là où le problème apparait. Retourner un «

pointeur » sur le code de la fonction anonyme ne suffit pas. Il faut aussi fournir des élements de son environnement qui feraient défault ultérieurement : les valeurs de ses variables libres. Celles qui ne font pas partie des paramètres de la lambda.

Et de plus, on a besoin que cette information survive à la dés-allocation de l'enregistrement d'activationde « parametricPredicate ». Pour celà, on va allouer une structure contenant valeur des variables libres de la lambda et pointeur sur son code sur le tas (heap).

C'est cette structure qu'on appelle « Closure » ou « Fermeture » : on fait l'inventaire des variables libres four « fermer » la lambda. En ce sens, on peut dire qu'une fonction anonyme liée lexicalement est un concept du langage et une « Fermeture » un détail d'implémentation. Après, peut-être à tort, les termes « lambda » et « closure » sont devenus interchangeables pour certains.

Et maintenant, lors de l'évaluation de « predicate(81) » on fera référence à la valeur de « c » lors la définition de la lambda et non celle de « main ». Encore une fois, c'est un choix qui s'est débattu et le consensus actuel veut que la liaison lexicale soit la bonne approche.

C'est d'ailleurs le monde de la programmation fonctionnelle, tout au long de la fin du 20e siècle, qui a poussé dans ce sens (avec ici deux emblématiques représentants, Haskell et OCaml), emportant dans son sillage le monde lispien. Ici, vous avez deux rapports décrivant la mise en œuvre de l'un et l'autre. Et dans les deux cas la notion de closure fait intégralement partie de la machine virtuelle (ou de l'environnement d'exécution) du langage.

Dans le cas d'OCaml, il y est dit qu'une valeur fonctionnelle est « évidemment » représenté pas une closure, c'est-à-dire une paire pointeur de code / environnement de liaison des variables. Il y est aussi expliqué qu'elles sont allouée dans une espace encore différent du tas pour de subtiles raisons de récupération mémoire.

Pour Haskell, comme le montre ce schéma, c'est un peu plus compliqué, ne serait-ce que du fait de choix de mise en œuvre d'un langage paresseux ! Mais si on y prête attention on retrouve encore le couple pointeur de code / environnement.

En plissant fortement les yeux, on notera tout de même une similitude entre les « Closures » et les objets des langages OO : une structure avec un dictionnaire de valeurs et un pointeur sur du code qui fait référence à ce dictionnaire. D'ailleurs Guy Steele dans ces « lambda the ultimate » papers en parle de manière indirecte.

En 2007, Rich Hickey après avoir fait une première tentative sur la plateforme .NET, propose un dialecte de Scheme ciblant la JVM. Et forcément, il adopte la liaison lexicale comme fondement pour la définition des fonctions anonymes.

Et bien d'autres partis pris qu'il détaille dans cet article des « Proceedings of the ACM on Programming Languages » (appels terminaux et récursivité).

Voyons comment il a (ou aurait) pu procéder pour mettre des « closure » sur une plateforme qui n'est pas initialement prévue pour.

On va d'abord parler d'une transformation classique utilisée par les compilateurs pour des langages fonctionnels. Elle n'est pas requise pour la suite, mais mérite qu'on la signale : Le « lambda lifting ». D'autant plus que dans la compilation de Java aujourd'hui, et pour une raison qui m'échappe, on l'appelle « lambda desugaring ». Il faudra qu'un jour, je pose la question à Rémi Forax...

L'idée est très simple : une première transformation du code fait apparaitre une fonction sans variable libre, au plus haut niveau, correspondant à la lambda que

l'on veut compiler

Forcément, elle est susceptible d'avoir plus de paramètres que la lambda d'origine, puisque les variables libres ne doivent plus l'être.

Là, vous me direz, on a fait que déplacer le problème... C'est pas faux !

En fait, il y a une seconde transformation, qui elle en revanche en fondamentale pour notre sujet : la « closure conversion ».

Et c'est là que la similitude en objet et closure se voit un peu mieux.

Sauf qu'ici, j'ai décidé d'utiliser une structure, plutôt qu'une classe, qui embarque la variable capturée...

... et le pointeur dur le code.

Ce qui permet de réécrire « parametricPredicate »...

... de la façon suivante : elle retourne une représentation de la fonction anonyme accompagnée des valeurs des variables capturées. C'est une représentation opérationnelle d'une valeur fonction.

Et forcément au point d'appel, il faut aussi effectuer une adaptation. On appelle le code de la fonction lambda liftée avec ces deux arguments, la valeur capturée et celle qu'on veut tester.

L'utilisation de fonctions anonymes fait son chemin dans bien des langages. Peut-être parce que de plus en plus de développeurs sont sensibilisés à la programmation fonctionnelle qui en use abondamment... Ou parce que ça fait bien, allez savoir ?

En attendant, la gestation du support des lambda dans Java débute à la sortie de Clojure, en 2007 et va quand même durer 7 ans, pour voir officiellement sortir la JSR 335 dans Java 8.

Pourquoi tant de temps ? Parce que Java est un langage orienté objets et que la somme des concepts sujets à la liaison lexicale dépasse les variables d'une fonction anonyme Lisp. Ce qui est très bien expliqué
dans cet article de Neil Gafter, membre d'un groupe ayant formulé une des trois propositions d'intégrations initiales (rien que ça). Mais aussi parce que les enjeux de rétro compatibilité et d'évolutivité de Java en tant que plateforme sont grands.

Voilà à quoi ressemble notre exemple en Java. J'ai utilisé ici la version 21 plutôt que la 8 avec la fonctionnalité classe anonyme et méthode 'main' d'instance pour qu'il ressemble plus à la version Kotlin.

La syntaxe pour les lambda est assez commune et proche de celle de Kotlin (aux accolades près).

En revanche, il n'y a pas de syntaxe pour décrire un type fonctionnel. Ici le type de retour de « parametricPredicate » est nominatif, c'est une interface.

Mais pas n'importe qu'elle interface. C'est une « Single Abstract Method » ou « SAM » interface...

... qui est annotée « @FunctionalInterface ».

Comme je l'ai déjà dit, une closure ressemble à un objet de disposant que d'une seule méthode pouvant accéder aux champs de l'objet. C'est cette similitude qui est exploitée ici. On assimile une certaine forme d'interface au type d'une lambda. On est là dans la recherche de rétrocompatibilité, pour exploiter
des interfaces déjà existantes qui auraient la bonne forme, comme « Runnable ».

Après cette forme de « SAM conversion » est aussi disponible en Kotlin, alors pour s'être privé d'une syntaxe explicite, je n'en sais rien.

En tout cas la stratégie de compilation retenue pour Java ressemble plutôt à ça.

On dispose toujours de notre fonction lambda-liftée... Enfin « desugared » dans la terminologie de Java.

Mais la traduction de la lambda lexicalement liée, la closure, s'appuie sur une classe créée localement à partir de l'interface fonctionnelle cible.

Elle comporte un champ pour stocker la valeur de la variable capturée et un constructeur pour la créer en fournissant la valeur à capturer.

Et d'une mise œuvre de la « Single Abstract Method » qui s'appuie sur la fonction anonyme lambda liftée. .

Et comme tout à l'heure, il ne reste plus qu'à créer et retourner la closure dans « parametricPredicate ».

Si c'est là bien le principe retenu pour les lambdas en Java, il est pourtant mis en œuvre avec une subtilité..

On va descendre un peu plus dans la soute et jeter un œil rapidement au bytecode généré par le compilateur, grâce à la commande « javap » sorte de dés-assembleur pour le code de la JVM.

Et en fouinant dans le résultat, une première chose qui saute aux yeux, c'est notre fameuse fonction anonyme lambda-liftée, ici appelé « lambda$parametricPredicate$0 ». On voit bien qu'elle a deux paramètres au lieu d'un et qu'elle retourne un booléen.

On retrouve aussi « parametricPredicate » qui doit retourner une valeur fonction de type « IntPredicate ».

Les concepteurs de la JSR 335 vont tirer partit de l'instruction de la JVM introduite en Java 7 pour les langages dynamique : « InvokeDynamic ». L'idée, c'est de se rendre résistants à de potentiels futurs choix de mise en œuvre en générant cette classe interne instance de « IntPredicate » à la volée.

Cette instruction référence une méthode Java dite d'amorçage (« Bootstrap ») décrite à l'index 0 d'une table des méthodes d'amorçage. L'idée, c'est que la méthode que l'on va appeler effectivement n'est pas connue avant l'exécution de cette méthode. Elle va littéralement construire le point d'appel d'une autre méthode, prenant un entier en paramètre et retournant un « IntPredicate », à la volée.

Et quand on cherche ladite table des méthodes d'amorçage, on trouve effectivement la description d'une méthode statique « metafactory » de la classe « LambdaMetafactory ».

Sans trop rentrer dans des détails qui dépassent le cadre de cette présentation, on voit qu'elle prend comme argument la fonction lambda-liftée « lambda$parametricPredicate$0 ». Et son travail va consister à construire une classe similaire à la classe « Closure » de mon exemple précédent, et à générer l'appel à son constructeur avec la valeur de « c » en paramètre. Cette méthode d'amorçage est écrite en Java et vous pouvez facilement en consulter le code qui se trouve dans la runtime. D'ailleurs pour générer cette classe, elle s'appuie sur un clone de la bibliothèque de manipulation de bytecode « ASM ».

Tout se passe comme dans le schéma suivant: l'instruction « invokedynamic » est exécutée dans le flux de bytecode...

... la méthode de bootstrap correspondante est invoquée ...

... qui produit une classe à la volée et « CallSite », ou point d'appel, pour son constructeur prenant en argument la valeur de la variable « c ».

Mais dans le cas des lambdas, ce call site est de type constant (personne ne viendra le modifier comme celà pourrait arriver dans un langage dynamique). Donc tout appel ultérieur sautera directement au « call site » et le JIT pourra s'en donner à cœur joie pour optimiser le tout.

Le fonctions anonymes liées lexicalement ont tellement fait leur chemin qu'on les retrouve dans deslangages sans processus de récupération automatique de la mémoire comme C++ ou Rust.

Dans le cas de Rust ça n'est pas particulièrement étonnant. Son concepteur initial, Graydon Hoare, fin connaisseur de la mise en œuvre des langages de programmation et qui se décrit comme un « langage engineer », est un amateur de d'OCaml. D'ailleurs les premières versions de Rust, avant qui ne soit « self hosted », étaient écrite en OCaml.

Et effectivement à regarder rapidement notre exemple fil rouge en Rust on n'est pas dépaysé !

La déclaration d'un type fonctionnel est à mi-chemin entre Kotlin et Java, et l'écriture d'une lambda en elle-même n'a rien de révolutionnaire.

Tout au plus les impératifs liés au modèle de gestion mémoire de Rust imposent quelques subtilités, comme ici l'usage du mot clef « move » pour gérer la capture de la variable « c » qui doit survivre à l'invocation de « parametric_predicate ».

Ce que l'on pourra retenir de tout ça c'est qu'une lambda, une fonction anonyme, peut être dynamiquement ou lexicalement liée. Et que dans le second cas pour gérer la capture de son environnement in faut une représentation des valeurs fonctionnelles que l'on appelle closure. Que les stratégies de mises en œuvre sont nombreuses et doivent tenir compte des spécificités sémantique du langage que l'on traduit.