Mémoire de recherche appliquée

Promotion : Architecture des Logiciels

Année : 2013-2014

Sous la direction de Nicolas Vidal

LES IMPACTS DE LINQ SUR UN PROJET .NET

Par Roman Leichnig et Pierre Tondereau

- École Supérieure de Génie Informatique -

Sommaire

I. Les besoins auxquels répond LINQ 5

1. Pourquoi LINQ ? 5

2. Naissance d’une solution 7

a. Les objectifs de base 7

b. Historique 8

c. Les domaines d’application de LINQ 10

II. Les apports de LINQ aux langages de .NET 17

1. Le typage implicite 17

2. Les initialiseurs d’objets 18

3. Les expressions lambda 20

4. Les méthodes d’extension 22

III. Technologies semblables 24

1. Scala 24

a. Présentation 24

2. Scala vs Java 25

a. Préparation 25

b. Test de scala par le code 25

c. Test de JAVA par le code 27

d. Conclusion 31

3. F# 32

4. F# vs C# 33

a. Introduction préliminaire 33

b. La Curryfication 33

c. L’algorithme du Tri Bulles 34

d. Conclusion 35

5. ProLog 36

IV. Les performances de la technologie 38

1. En environnement Windows 38

2. En environnement MacOs 38

Introduction

Le métier de développeur se résume bien souvent à utiliser du code pour traiter des données dans le but de produire une solution logicielle à un problème donnée. Si, le résultat final doit être simple d’utilisation, sa réalisation peut s’avérer des plus complexes. Pour réaliser un logiciel, il faut écrire du code. Ceci peut être fait dans différents langages de programmations qui varient selon le contexte de l’entreprise, les préférences des développeurs, les compétences des équipes de développement ou encore le système d’exploitation visé.

Quel que soit le langage choisi, on sera toujours confronté à des problématiques de traitement des données, que celles-ci proviennent fichiers textes, d’une base de données ou encore de fichiers XML, voire de tout cela à la fois. Ces formes de traitement étant communes à tout le monde du développement logiciel, nous en sommes en droit d’attendre de trouver dans nos outils de développement des moyens simples de les gérer. C’est dans ce contexte que s’inscrit la technologie LINQ : elle est une plateforme d’accès aux données communes aux langages du .NET Framework visant à faciliter la vie du développeur par le biais d’un large panel de méthodes simples permettant l’accès et le traitement de données.

En considérant que l’on travaille sur un projet reposant sur .NET, il y a de très fortes chances pour que, à un moment donné, on ait besoin d’interroger une base de données et d’affecter les réponses de la base à des objets. Le problème qui se pose ici est que bien souvent, quand on utilise une base de données relationnelle, il existe un écart gigantesque entre la base de données et le langage de programmation que l’on utilise. La motivation initiale de LINQ était donc d’aborder les difficultés conceptuelles et techniques rencontrées lors de l’utilisation d’une base de données relationnelle au sein d’un projet .NET. Cette technologie vise donc à fournir une solution pour le problème de mapping objet-relationnel, ainsi qu’à simplifier l’interaction entre les objets et les sources de données. Au fil du temps, LINQ a évolué vers un ensemble d’outils destinés à l’interrogation et le traitement des données intégrés dans le .NET Framework. Il permet d’accéder aux données par le biais d’objets stockés en mémoire (LINQ to Object), d’interroger une base de données (LINQ to SQL), des documents XML (LINQ to XML) ou encore un système de fichiers.

En somme, LINQ se revendique comme étant la passerelle entre le monde des données et le monde de la programmation générale. Il unifie l’accès aux données, peu importe la source. Il permet un niveau de précision équivalent à celui fourni par SQL en intégrant bon nombre de fonctions similaires (comme WHERE, GROUP BY ou encore ORDER BY). Ainsi, LINQ offre la possibilité d’exécuter des requêtes à l’intérieur même d’un langage de programmation ; d’où son nom *Language INtegrated Queries*. Un des aspects clés de LINQ est qu’il a été conçu pour être utilisé sur n’importe quel type d’objet ou source de données et peut fournir un modèle de programmation cohérent pour le faire. La syntaxe et les concepts sont les mêmes dans toutes ses utilisations : si on apprend comment attaquer une collection d’objets ou un tableau avec LINQ, on saura à 90% comment l’utiliser avec une base de données ou un fichier XML.

Enfin, LINQ présente le gros avantage de n’être utilisable qu’en environnement fortement typé. Il permet donc de bénéficier de la vérification des requêtes avant compilation ainsi qu’une écriture largement facilité par l’*IntelliSense* proposée par *Visual Studio*. LINQ est une réelle avancée vers un modèle de programmation plus déclaratif qui change radicalement notre façon d’accéder et de gérer les données et qui, de surcroit, diminue considérablement la quantité de code d’une application.

Au cours des chapitres à venir, nous nous efforcerons d’énumérer et d’analyser les impacts que peut avoir l’utilisation de la technologie LINQ au sein d’un projet bâti sur le .NET Framework. Pour ce faire, nous commencerons par expliquer en détail ce à quoi correspond et comment est articulée cette technologie. Ensuite, nous examinerons les technologies semblables proposées par les concurrents de Microsoft pour continuer par avec une mise en exergue des performances de LINQ dans différents environnement. Une fois ces points exposés, nous tenterons d’apporter à une réponse aux questions posées par l’utilisation de ce produit grâce à l’avis de professionnels coutumiers de celui ci.

# Les besoins auxquels répond LINQ

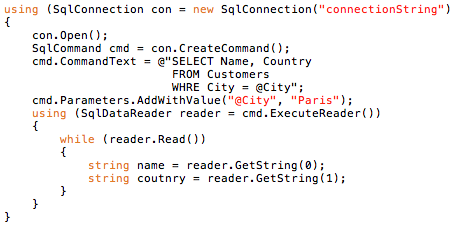
## Pourquoi LINQ ?

Un constat simple est à l’origine de la technologie LINQ : la grande majorité des applications qui sont développées utilise des données provenant d’une base de données relationnelle. De ce fait, l’apprentissage d’un langage comme C# n’est pas suffisant pour couvrir l’ensemble du domaine technique nécessaire à la réalisation d’une application. Il faut aussi apprendre d’autres langages, comme SQL et également une API permettant de le lier à C# pour réaliser une application complète. Afin de démontrer l’utilité de la technologie, nous allons utiliser un exemple d’accès aux données utilisant la méthode classique utilisée en .NET. Nous verrons ainsi que LINQ vise à combler un manque de cohérence entre les sources de données et les langages de programmation.

L’utilisation fréquente des bases de données dans les applications reposant sur .NET nécessite l’utilisation d’API pour accéder aux données. Ainsi, le Framework embarque ADO.NET, qui fournit un ensemble de classes destinées à représenter les données d’une base relationnelle en mémoire. Cette API se compose de classes telles que *SqlConnection*, *SqlCommand*, *SqlReader, DataSet*  ou encore *DataTable*. Le problème de ces classes est qu’elles forcent le développeur à travailler explicitement avec des tables, des enregistrements et des colonnes alors que les langages modernes tels que C# ou VB.NET utilises des paradigmes orientés objets.

Maintenant que l’objet est le modèle prédominant dans le monde du développement logiciel, les développeurs subissent de grandes difficultés dans le domaine du mapping entre objets et base de données relationnelles. Le résultat est que beaucoup de temps est consacré à résoudre ce problème et que cela a de lourdes conséquences sur la productivité au sein d’un projet IT. En plus du gain en productivité induit par la technologie LINQ, on améliore également la qualité du code produit.

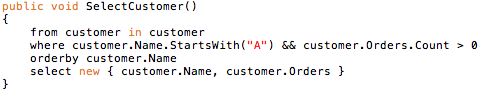
Exemple d’accès aux données classiques en C# :



On notera que les classes utilisées dans cet exemple sont dédiées au moteur de SQL Server et ne peuvent être utilisées avec aucun autre moteur de base de données.

Il existe d’autres possibilités que d’écrire ces instructions à la main. On pourrait très bien utiliser un générateur de code ou l’un des nombreux outils de mapping objet-relationnel disponibles. Le problème est que ces outils ne sont pas parfaits non plus et ont leurs propres limites. Par exemple, s’ils sont conçus pour accéder à des bases de données, il ne traitent, la plupart du temps, qu’avec ce type de sources. L’idée principale est donc que, en utilisant LINQ on sera en mesure d’accéder à n’importe quelle source de données en écrivant des requêtes simples, comme nous allons le monter dans l’exemple suivant, dans un langage que l’on maîtrise et que l’on utilise au quotidien.

Accès aux données en utilisant LINQ :



Dans cette requête, les données pourraient être dans la mémoire, dans une base données ou dans un fichier XML, la syntaxe resterait semblable, sinon exactement la même chose.

Notons aussi que le fait que les développeurs d’applications soient confrontés à plusieurs langages de programmation, des bases de données relationnelles, des documents XML et tout le reste signifie qu’ils ont besoin de deux choses :

* Etre capables de travailler avec chacune de ces technologies individuellement
* Et aussi être capables de les mélanger pour travailler à des solutions riches

Le problème que pose ce constat est que la programmation orientée objet et le modèle de base de données n’ont jamais été conçus pour travailler ensemble. Il représente tous les deux des paradigmes différents qui s’accordent difficilement.

Les données sont généralement manipulées par l’application écrite en utilisant la programmation orientée objet par le biais de langages tels que C#, VB.NET, Java. Delphi ou encore C++. Mais traduire un graphe objet en une autre représentation, comme un ensemble de base de données nécessite souvent un travail long et fastidieux. Le problème général à l’origine de la création de LINQ et qu’il est destiné à solutionner est donc décrit par Microsoft comme : « *Data != Objects ».* Si l’on compare l’objet au paradigme relationnel, les différences existent à de multiples niveaux, en voici quelques-unes :

* Les bases de données et les langages orientés objets ne partagent pas le même ensemble de types de données primitifs. Par exemple, les chaînes ont généralement une longueur délimité dans les bases de données, ce qui n’est pas le cas en C#. Cela peut être un problème si l’on essaie de persister une chaine de 150 caractères dans un champ de table qui accepte seulement 100 caractères. Un autre simple est que la plupart des bases ne recense pas de type booléen, alors que les valeurs vrai / faux sont constamment utilisées en programmation.
* La programmation orientée objet et les théories relationnelles sont livrées avec différents modèles de données. Pour des raisons de performance et en raison de leur nature intrinsèque, les bases de données relationnelles sont généralement normalisées. La normalisation est un processus qui élimine la redondance, organise efficacement les données et réduit le risque d’anomalie au cours des opérations sur les données. Cela empêche un mapping direct des tables à des objets et des collections.
* Avec SQL, nous avons la possibilité d’écrire des requêtes de manière déclarative, ce qui permet d’atteindre toutes les données qui nous intéresse en une seule fois. Inversement, un langage de programmation impérative comme C# ou VB.NET force à utiliser une boucle pour récupérer un jeu de données.
* Il existe également une différence au niveau de l’encapsulation entre les bases de données relationnelles et le paradigme objet. En effet, un objet est prévu pour contenir des données ET des comportements alors qu’un enregistrement dans un modèle relationnel ne comprend que des données. On peut agir sur un enregistrement par le biais de requêtes ou procédures stockées mais pas directement. Le modèle relationnel offre donc une séparation entre le code et les données, à l’inverse du paradigme objet.
* Certains concepts ne sont pas pris en charge par les bases de données relationnelles. Par exemple, il est impossible de représenter l’héritage dans ce type de modèle. Ainsi, si l’on se trouve en présence d’une clase héritant d’une autre, on ne pourra pas représenter cette relation dans les deux modèles.

## Naissance d’une solution

### Les objectifs de base

Au démarrage du projet, LINQ a était destiné à remplir 9 objectifs principaux :

1. *Unifier les requêtes*

Microsoft a eu la volonté de créer une syntaxe de requête unifiée entre les différentes sources de données pour éviter d’avoir à employer un langage spécifique pour chaque type de données.

1. *Offrir un langage d’interrogation à C# et VB.NET*

LINQ a pour but de d’intégrer aux langages du .NET Framework les capacités d’interrogations de langages comme SQL ou XQUERY.

1. *S’élargir aux autres langages de programmation*

Il est intéressant de noter que dès le lancement du projet LINQ, Microsoft avait la volonté de l’étendre à ses autres langages de programmation (autre que C# et VB)

1. *Atteindre le plus de sources de données possibles*

LINQ a pour vocation de proposer un accès à d’autres sources de données que les bases de données relationnelles ou les documents XML.

1. *Etre fortement typé*

Le fait de proposer une technologie fortement typée permet de prévenir les erreurs classiques rencontrées dans la programmation avec des langages à faible typage (type PHP). En effet, le compilateur détecte instantanément les erreurs dans les requêtes, sans avoir à les exécuter.

1. *Offrir l’IntelliSense*

Grace à LINQ, qui s’emploie au sein de C# ou VB, et donc dans Visual Studio, Microsoft permet aux développeurs d’écrire des requêtes en profitant pleinement de son *IntelliSense* (auto-complétion extrêmement puissante) et donc de gagner en vitesse d’écriture et productivité.

1. *Support de débogage*

La technologie LINQ est débogable au sein de Visual Studio au même titre que n’importe quelle autre portion de code C#. On peut ainsi suivre au pas à pas l’exécution d’une requête LINQ.

1. *Utiliser les bases construites dans les versions précédentes*

Microsoft a fait en sorte de conserver le meilleur C# 2.0 et VB.NET 8.0 pour l’injecter dans la technologie, tout en épurant tout l’obsolète.

1. *Rester 100% rétro compatible*

Microsoft s’est penché sur la rétro compatibilité du produit en lui permettant de travailler sur les collections classiques et génériques, le data-binding, les contrôles issus de Windows Form, etc.…

La caractéristique principale de LINQ à retenir est qu’il est fortement typé. Cela permet :

* De vérifier toutes les requêtes lors de la compilation. Contrairement au instructions SQL, ou nous ne nous maitrisons pas forcément toute les erreurs potentielles avant l’exécution, nous pouvons vérifier nos instructions LINQ au fil de l’eau, après l’ajout de chaque clause par exemple, ou d’une colonne supplémentaire.
* De bénéficier des possibilités proposées par l’IntelliSense de Visual Studio lors de l’écriture de requêtes LINQ. Cela permet non seulement de taper plus vite mais aussi de travailler avec des requêtes très complexes en réduisant la difficulté de leur écriture.

### Historique

Initié par des projets comme XLinq ou DLinq (respectivement ancêtres de LINQ to XML et LINQ to SQL), LINQ est le résultat d’un processus de d’expérimentations et de travaux sur le long terme au sein du département Recherche et Développement de Microsoft. Il est un formé d’un regroupement de plusieurs projets impliquant des évolutions des langages de programmation et des méthodes d’accès aux données initialement présents dans le Framework .NET.

Le projet Cω

Le langage Cω (C-Omega) est un projet de Microsoft Research qui a étendu le langage C# dans plusieurs domaines, notamment les suivants :

* Une extension de contrôle de flux asynchrone (connue sous le nom de Polyphonic C#)
* Un type de donnée destiné à la manipulation d’objets XML et de base de données (connu respectivement comme Xen et X#)

Cω ne couvrait pas un domaine de compétence aussi large que celui couvert par LINQ, mais une grande partie de ce qu’est maintenant ce produit était déjà présent dans Cω. Le projet Cω a été conçu pour expérimenter des requêtes intégrées au langages de programmation, en mélangeant C# et SQL, C# et XQuery, etc.… Fondé par des chercheurs tels que Wolfram Schulte, Erik Meijer et Gavin Bierman, le projet Cω est sorti en version beta en 2004. Ce prototype a apporté énormément à ce que deviendrait plus tard le projet LINQ et, quelques mois après sa sortie, Anders Hejlsberg, le concepteur en chef de C#, a annoncé que Microsoft travaillerait désormais à intégrer le maximum des connaissances apportés par Cω à C# et ses autres langages de programmation. Anders ajoute, à ce moment, qu’il a passé la grande majorité des deux dernières années à s’interroger sur les disparités existantes entre les langages de programmation, C# en particulier, et le monde des données. Le monde des données auquel il fait référence comprend, évidemment SQL, mais aussi XML et XQuery. Ainsi, les extensions de Cω injectées dans .NET et plus particulièrement C# ont été les premières étapes d’un système unifié traitant les requêtes SQL, les jeux de résultats et le contenu d’un document XML en tant que membre à part entière du langage. De plus, Cω à introduit le type Stream qui correspond au type System.Collections.Generic.IEnumrable de la version 2.0 du Framework .NET. Aux autres apports de Cω, on notera les constructeurs pour tuples typés, ainsi que le support d’XML embarqué dans le projet.

Le projet ObjectSpaces

Bien avant d’envisager la création de LINQ to SQL, Microsoft s’était déjà penché sur le problème du mapping objet-relationnel, et ce dès 2001 avec le projet ObjectSpaces. Apparu pour la première fois lors d’une présentation de ADO.NET en 2001, ObjectSpaces était un ensemble de module destiné à fonctionner au sein de l’API. Ce produit permet, pour la première fois avec les technologies Microsoft, de manipuler les données comme des objets, de manière indépendante de la source de données. En 2004, Microsoft a annoncé que ObjectSpaces dépondait du projet WinFS5[[1]](#footnote-1) et, de ce fait, serait reportée à la période Orcas (soit les prochaines versions de .NET, à savoir 2.0 et de Visual Studio, la 2005). Aucune autre version n’est parue après cette annonce. Tout le monde compris que ObjectSpaces ne verrait jamais le jour lorsque Microsoft annonça que WinFS ne serait pas intégré à la première version de Windows Vista.

Implémentation de XQuery

Pour servir les mêmes objectifs que le projet ObjectSpaces et à la même période, Microsoft avait commencé à travailler sur un moteur XQuery. Un aperçu de son moteur avait été inclus dans la version beta de .NET 2.0, mais la firme à finalement décidé de ne pas propulser une implémentation *client-side* de XQuery (à l’inverse de l’implémentation *server-side* proposée par SQL Server 2005) dans la version finale. Ce rejet de l’implémentation était principalement imputable, selon Microsoft, au fait que XQuery représente un langage supplémentaire à appréhender spécifiquement pour la manipulation d’objets XML.

Ainsi, au terme de plusieurs années de recherche, ponctuées par l’échec de différents projets sur le plan individuel (à savoir ObjectSpaces, l’intégration XQuery à .NET et Cω), Microsoft à finalement annoncé son dernier né, le projet LINQ, lors de la Professional Developer Conference (PDC) de 2005.

Conçu par Anders Hejlsberg et d’autres grands noms de la maison de Seattle, l’objectif principal de projet LINQ était de remédier au problème récurrent des incompatibilités entre paradigmes objet et relationnel à partir de langages comme VB.NET et C#. Avec LINQ, nous pouvons requêter sur à peu près n’importe quoi, grâce à des expressions simples intégrées aux langages de programmation.

La première version de LINQ est livrée avec la version 2008 de Visual Studio et la mise à jour 3.5 du Framework .NET. Cette version de .NET est livrée avec des librairies complémentaires ainsi que des compilateurs C# et VB remis au gout du jour, mais elle reste compatible avec la version 2.0.

Il est essentiel de comprendre que, bien que C# et VB ait été enrichi et que plusieurs librairies aient été ajoutées à .NET, la CLR (le runtime .NET) n’a pas besoin d’évoluer. C’est cette utilisation de l’ancienne version de la CLR qui permet à LINQ sa rétrocompatibilité avec .NET 2.0.

### Les domaines d’application de LINQ

La technologie LINQ est destinée à couvrir la couche d’accès aux données en fonction de la source de données. Elle permet, en effet, de manipuler des objets (LINQ to Object), des fichiers XML (LINQ to XML), un jeu de données SQL (LINQ to SQL), ou encore un modèle mappé au sein du projet (LINQ to Entities). Au cours des paragraphes à venir, nous nous efforcerons d’expliquer le fonctionnement de ces modules par le biais d’exemple d’utilisation.

#### LINQ to Object

Comme annoncé précédemment, LINQ permet la manipulation d’objets, l’exemple suivant est destiné à exposer la syntaxe à employer pour exécuter une requête LINQ sur une collection d’objets :

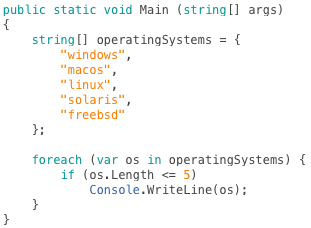
#### 

Une fois compilé, ce morceau de code produit en sortie :

macos

linux

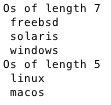
Comme le montre les résultats, cette requête LINQ vise à filtrer la liste *operatingSystems* en ne sélectionnant que les mots dont la longueur est inférieure ou égale à 5. Nous pourrions dire sans difficulté que le même résultat peut être obtenu sans utiliser LINQ, mais en écrivant du C# classique :



Il est même évident, en voyant le morceau de code « classique », que cette solution est beaucoup plus courte que la version LINQ et très facile à lire, de surcroit. En revanche, si l’on introduit les notions de regroupement et de tri, il est beaucoup plus facile de percevoir l’intérêt présenté par LINQ :



Cet exemple fournit la sortie suivante :



Nous avons donc exprimé dans cette requête (ou deux requêtes imbriquées plus précisément) ce qui serait traduit en français par « trier une liste de mots par ordre alphabétique et les regrouper par leur longueur dans l’ordre décroissant ». L’un des premiers avantages de LINQ que démontre cet exemple est qu’il permet d’exprimer de manière déclarative ce que nous voulions réaliser en utilisant des requêtes plutôt qu’en écrivant des morceaux de code alambiqués.

En plus des mots clés typiques de SQL visible dans ces exemples, LINQ fournit également un certain nombre d’autres fonctions telles que Min, Max, Avg, Sum, etc.…

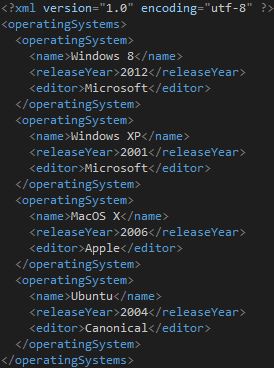
Un exemple d’utilisation pourrait être :



Nous reviendrons en détail sur l’utilisation du symbole « => » dans le chapitre des apports aux langages de programmation consacrés aux expressions lambda.

#### LINQ to XML

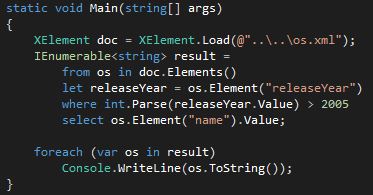
LINQ peut également être utilisé sur des fichiers XML. Il se présente comme une API d’interrogation XML complète, au même titre qu’un System.Xml sous .NET 2.0 (en modernisé) et possède quelques caractéristiques fondamentales de XPath et XSLT. LINQ to XML permet de charger en mémoire un document XML et de travailler dessus, que ce soit en édition ou pour la lecture de son arbre d’éléments. On pourra donc employer LINQ to XML pour effectuer facilement un grand nombre de tâches liées au traitement de fichiers XML.



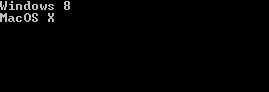
Soit le fichier XML suivant :

Supposons que nous voulons afficher la liste des systèmes d’exploitation dont la sortie est ultérieure à l’année 2005.

Voici la méthode à appliquer avec LINQ to XML :



Ce qui produit le résultat suivant :



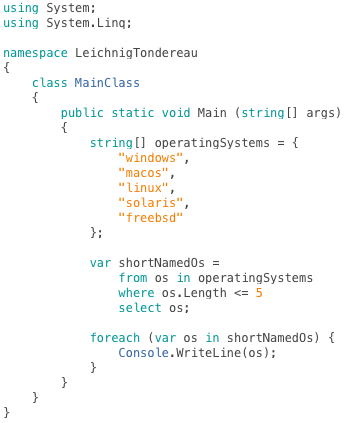
Comme le montre l’exemple, la technologie LINQ permet au développeur d’interroger un fichier XML de la même manière que s’il interrogeait une collection d’objets. On retrouve en effet les mots clés apportés par LINQ à C# tels que from, where et select. L’intérêt de LINQ démontré par cet exemple est donc l’aspect unifiant de la technologie, qui permet d’employer le même langage sur différentes sources de données.

#### LINQ to SQL

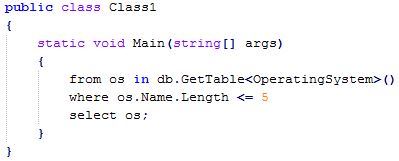
Dans la continuité de la notion d’unification exprimée dans le paragraphe consacrée à LINQ to XML, il est important de rappeler que Microsoft a voulu, en créant en LINQ, faire des requêtes une brique à part entière et naturelle de ses langages de programmation. Il était donc naturel d’intégrer au projet LINQ une couche d’accès aux données stockées dans une base de données relationnelle. C’est cette couche qui a été baptisé LINQ to SQL.

LINQ to SQL est basé sur une extension d’ADO.NET qui permet de mapper les tables et les lignes à des classes et des objets. Une table devient donc une classe et chaque colonne de cette table devient une propriété de la classe tandis que les relations entre les tables sont matérialisées par des propriétés supplémentaires.

Reprenons donc l’exemple traité pour illustrer l’utilisation de LINQ to Object (liste de systèmes d’exploitation triée en fonction de la longueur des chaines de caractères) :



Et voici la syntaxe pour requêter dans une base relationnelle :

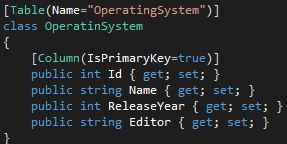


Cette requête LINQ est donc destinée à rapatrier une liste de système d’exploitation depuis une base de données. La requête est donc la même que si on extrayait les données depuis une collection à la différence prêt de l’objet sur lequel on travaille. En effet, dans le premier cas nous utilisons *operatingSystems*, qui est une collection de String alors que pour la requête SQL nous faisons appel à une méthode *GetTable().* La puissance de LINQ se traduit ici par le fait que le moteur va transformer notre requête LINQ en une véritable requête SQL, qui est le seul langage pouvant être interprété par une base de données relationnelle. Une fois encore, cela permet au développeur de bénéficier de tous les avantages induits par un langage fortement typé, là où il aurait dû affronter les potentielles erreurs de syntaxes récurrents avec l’utilisation de SQL, qui n’est vérifié qu’à son exécution.

Afin de comprendre au mieux le fonctionnement de LINQ to SQL, il est essentiel d’expliquer ce à quoi correspond l’instruction *db.GetTable<OperatingSystem>()* :

Afin d’assurer le mapping objet-relationnel, Microsoft fournit avec .NET le produit EntityFramework qui permet de décrire une base de données relationnelle sous la forme de classe C# ou VB.NET.

Entity permet donc de déclarer des classes qui correspondront aux tables présentes dans la base de données. Cela s’effectue comme suit :



Ici, l’attribut *Table* de la classe permet d’identifier la table par son nom au sein de la base de données. Il est également important de noter que l’on doit spécifier la colonne qui correspond à la clé primaire de la table par l’attribut *IsPrimaryKey*. D’autres attributs sont également applicables sur une colonne, en voici une liste non exhaustive :

* *Name* : permet d’indiquer le nom de la colonne en base dans le cas où la propriété ne porte pas ce nom dans le code.
* *CanBeNull* : détermine si l’on traite une colonne nullable ou non.
* *IsDbGenerated* : détermine si la valeur de la colonne est générée automatiquement par la base.

Pour pouvoir utiliser LINQ to SQL correctement, il est impératif de déclarer un objet *DataContext* issu de du *namespace System.Data.Linq.*. C’est l’objet *DataContext* qui est chargé de faire le lien entre le code C# et la base relationnelle. Il se déclare en chargeant en mémoire un fichier d’extension *.mdf*, qui est la représentation physique d’une base de données. On le déclare comme suit :



Une fois le *DataContext* déclaré, il devient évident de l’utiliser comme source de données et donc d’exécuter des requêtes dessus. On accèdera donc à ses tables en utilisant l’instruction :



Il est important de noter que *DataContext.GetTable* est une méthode générique qui permet au développeur de travailler avec des objets fortement typés.

# Les apports de LINQ aux langages de .NET

Avec l’avènement de .NET 2.0, les langages de programmations de Microsoft se sont vus enrichis de nombreuses de nouvelles fonctionnalités, qui sont essentielles au fonctionnement de LINQ. A liste de ces fonctionnalités qui ont rendus LINQ possible, il on retrouve :

* **Les types implicites de variables**, ou l’utilisation du mot clé *var* comme type lors de la déclaration d’une variable. Cela est possible grâce à une déduction du type de la variable par le compilateur depuis l’expression utilisée pour l’initialiser.
* **Les initialiseurs d’objets**, qui permettent au développeur d’initialiser un objet directement après l’appel de son constructeur, en initialisant ses propriétés entre accolades ‘{ }’.
* **Les expressions lambda**, qui sont une évolution des méthodes anonymes.
* **Les méthodes d’extension**, qui permettent au développeur de déclarer des méthodes s’appliquant directement sur des types primitifs. Par exemple, on pourrait déclarer une méthode *EstFerie()* en extension à la classe *DateTime*. Cette méthode s’appellerait de la manière suivante : *myDat*e.*EstFerie()*.
* **Les types anonymes**, qui sont des types dynamiques permettant d’instancier des objets par le biais d’initialiseur d’objets

## Le typage implicite

Le large lot d’améliorations amenées par .NET 2.0 inclut l’apparition du mot clé *var*. Celui-ci permet au développeur de déclarer une variable sans avoir à spécifier explicitement son type.

Ainsi, la déclaration, qui auparavant s’écrivait nécessairement comme suit :



Peut dorénavant être écrite de cette manière :



Si, au premier abord, ce changement dans la syntaxe de déclaration des variables peut paraître anodin, il témoigne en réalité des profonds changements apportés au compilateur par la version 2.0 de .NET. En effet, celui-ci est dorénavant capable d’interpréter l’expression qui value une variable pour en déduire le type. Dans l’exemple proposé, le compilateur détermine donc que *value* est de type *int* en analysant la valeur 255. On pourra donc appliquer tout ce qu’il est possible d’appliquer à une variable de type *int* à *value*. Il est essentiel de souligner que les deux écritures sont strictement identiques au point de vue du compilateur : elles produiront exactement le même code IL *(Intermediate Language[[2]](#footnote-2)*) une fois compilées.

Contrairement à ce que cela pourrait laisser penser, les types anonymes impliquent, tout comme les langages où ils sont utilisés, un typage fort. Il sera doc impossible d’effectuer un transtypage d’entier vers chaîne, comme dans l’exemple suivant :



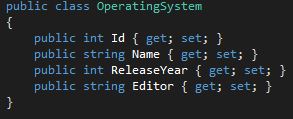
On voit très nettement dans cet exemple que Visual Studio identifie l’assignation de *value* à une chaine de caractères comme une erreur de syntaxe. Il y associe l’erreur « *Cannot Implicitly convert type ‘string’ to ‘’ ‘int’ »*, ou, littéralement, « Impossible de convertir une chaine en entier ».

## Les initialiseurs d’objets

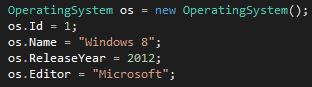
Tout comme les types anonymes, les initialiseurs d’objets sont apparus avec la version 2.0 de .NET. Ils consistent à initialiser les données membres d’une classe après l’appel de son constructeur sans le faire sur l’instance qui vient d’être créée.

Voici un exemple résumant la situation :

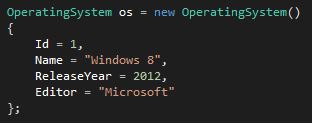
Soit une classe *OperatingSystem* définie de la manière suivante :



Avant l’apparition des initialiseurs d’objets, l’instanciation d’un objet de type *OperatingSystem* (sans constructeur paramétré) se faisait de la sorte :



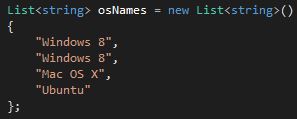
L’apparition de cette fonctionnalité a permis l’écriture de ce type de portion de code :



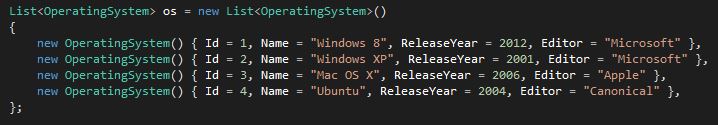
On voit ici qu’il est désormais possible d’initialiser les données membres d’une classe juste après son instanciation grâce à l’utilisation des initialiseurs d’objets qui se manifestent par l’initialisation de valeurs entre accolades ({ }).

Nous noterons également que les initialiseurs d’objets s’appliquent également aux collections. En effet, il est désormais possible d’ajouter dynamiquement des éléments à une liste, juste après sa déclaration sans passer par sa méthode *Add()*.

Exemple :



Une fois ce mécanisme assimilé, il possible de combiner les initialiseurs d’objets et de collections très simplement. Par exemple, nous pouvons aisément initialiser une collection d’objets de type *OperatingSystem* qui seront eux-mêmes initialisés dynamiquement dans l’initialisation de la collection :



Il est évident que cette nouvelle fonctionnalité dans les langages de .NET offre au développeur une très grande fluidité dans l’écriture de son code et permet de réduire considérablement les quantités d’instructions écrites.

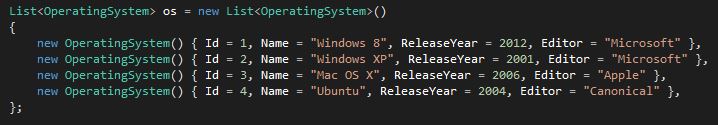
Tout comme dans le cas des types anonymes, l’initialisation d’un objet ou d’une collection, utilisant, ou non, un initialiseur d’objets produira exactement le même code IL à la compilation.

### Les expressions lambda

Les expressions lambda, qui sont la plus grande avancée dans C# vers un langage fonctionnel, sont issus du monde du lambda-calcul. Wikipédia définit le lambda-calcul comme étant « un système formel inventé par Alonzo Church dans les années 1930, qui fonde les concepts de fonction et d’application. Il a été le premier formalisme utilisé pour définir et caractériser les fonctions récursives et a donc une grande importance dans la théorie de la calculabilité, à l’égal des machines de Turing et du modèle de He brand-Gödel. » On notera que cette fonctionnalité n’est pas introduite par C# puisqu’elle est apparue auparavant dans des langages fonctionnels, tels que LISP.

A proprement parler, les expressions lambda fournissent au développeur une syntaxe concise permettant de réaliser ce qu’il peut déjà faire grâce à l’utilisation de méthodes anonymes, dont l’écriture s’avère très verbeuse et bien souvent fastidieuse. Prenons un exemple :

Soit une liste d’instances de notre classe *OperatingSystem*, utilisée dans les exemples précédents :

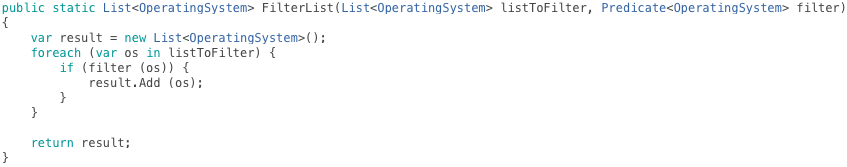


Si l’on voulait créer une seconde liste d’instances de *OperatingSystems* en filtrant la première de manière à ne sélectionner que les objets dont la propriété *Editor* est « Microsoft » en utilisant des méthodes anonymes, on procéderait de la sorte :

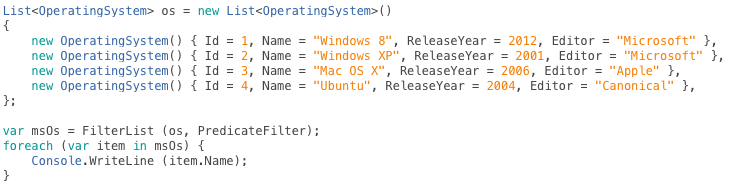
* D’abord, définir une méthode de filtrage, où l’on définit le filtre à mettre en place. C’est cette méthode qui sera utilisé comme *delegate* :



* Ensuite, définir fonction prenant en paramètre le *delegate* de tri et la liste à trier qui se verra appliquer le filtre :



* Puis appeler cette dernière dans notre méthode principale afin d’assigner son résultat à un objet sur lequel on pourra ensuite travailler :



Ce code produira en sortie le résultat suivant :



Voici maintenant la façon d’obtenir précisément le même résultat en utilisant la syntaxe LINQ et une expression lambda :



Cette portion de code, qui est environ 5 fois moins conséquente que si l’on avait utilisé des méthodes anonymes et autres *delegates*, produit exactement le même résultat que l’enchaînement de méthodes exposé dans l’exemple précédent, soit :



Intéressons nous maintenant à la structure d’une expression lambda :

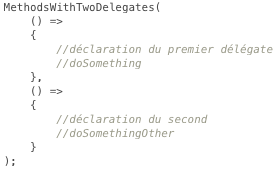


Paramètres d’entrée : c’est une représentation de l’objet sur lequel nous travaillons. Il est appelée « o » dans cet exemple mais peut prendre n’importe quel nom. Il est conseillé d’utiliser des noms explicites afin de faciliter la lecture des expressions lambda.

Opérateur : Au même titre que le l’opérateur « + » dans une adition, le « => » d’une lambda est ce qui lui donne son sens. Avec lui, on peut traduire une expression lambda par « les paramètres d’entrée remplissant les conditions énoncées par la condition à évaluer ».

La condition à vérifier : c’est elle qui est la clé de la lambda. Elle est le pivot sur lequel le compilateur va s’appuyer pour déterminer si l’objet passé en paramètre correspond à l’expression lambda en cours d’exécution.

On notera également que les expressions lambda utilisent les types anonymes qui permettent au développeur de ne pas spécifier le type du paramètre d’entrée. Celui-ci sera déduit par le compilateur en évaluant la condition placée à droite de l’opérateur. Il est aussi important qu’il est possible d’utiliser des expressions lambda pour déclarer des méthodes de manière anonyme. Par exemple pour passer deux *delegate* en paramètre d’une méthode, on pourra très bien utiliser la syntaxe suivante :



## Les méthodes d’extension

Comme nous l’avons énoncé précédemment, les méthodes d’extension permettent d’ajouter des méthodes à un type après qu’il ait été défini. De cette manière, il est dorénavant possible d’appliquer des méthodes sur les types primitifs où les classes du Framework comme si elles faisaient partie intégrante de leur déclaration. Pour illustrer ceci, prenons l’exemple d’une fonction retournant un booléen qui détermine si un nombre est pair ou non :

La méthode, sans passer par une extension, serait très certainement déclarée de cette manière :



Et appelée comme suit :



L’intérêt des méthodes anonymes est de pouvoir appeler une méthode directement sur un type que l’on n’a pas défini, exactement comme si cette méthode appartenait à sa déclaration d’origine. Ainsi, elles permettent d’appeler la méthode *isEven()* de la sorte :



Pour ce faire, il faudra déclarer la méthode *isEven()* comme suit :



Si l’on s’intéresse de près à la déclaration de cette méthode, on remarque qu’elle prend un paramètre en entrée alors qu’aucun paramètre ne lui est passé lors de son appel. C’est là la clé des méthodes d’extension : le premier paramètre à donner dans leur déclaration est un paramètre du type de la classe sur laquelle on veut appliquer la méthode, précédé du mot clé *this*. C’est précisément ce mot clé qui permet au compilateur d’identifier cet méthode comme étant une méthode d’extension et de nous autoriser à l’appeler directement sur le type qu’elle vise. On notera qu’il est impératif de déclarer une méthode d’extension comme *static* et dans une classe spécialement prévue à cet effet pour qu’elle soit utilisable.

Bien que la technologie LINQ n’ait pas inventé les fonctionnalités que nous venons d’énumérer, il est essentiel de comprendre qu’elle repose entièrement sur ces chacune des ce cinq notions, sans différence d’importance.

# Technologies semblables

## Scala

### Présentation

Scala est un langage récent, à typage statique, qui compile vers du bytecode Java et qui tourne sur la JVM (il peut aussi être compilé en MSIL et utiliser tout le framework .Net mais, étant donné que je n'utilise aucun des langages de Microsoft, ceci ne sera pas abordé ici). Son nom est la concaténation de l'expression anglaise « Scalable Language », qui veut dire langage évolutif ou « langage qui peut être mis à l'échelle ».

Il a été conçu à l'École Polytechnique Fédérale de Lausanne (EPFL) pour être un langage polyvalent, puissant et adapté aux petits scripts comme aux grands logiciels.

Martin Odersky, son créateur, est un professeur de programmation à l'EPFL. Il a longuement cherché à unifier la Programmation Orientée Objet et la Programmation Fonctionnelle. Le sujet de ses recherches était que ces deux paradigmes sont deux faces différentes d'une même pièce. Afin de prouver ses propos, il a essayé de créer plusieurs langages comme Pizza, GenericJava (une partie de ce langage a été fusionnée avec Java pour donner les classes et méthodes génériques de Java 5) et Functional Nets. Il a commencé le développement de Scala en 2001 et a publié la première version stable vers la fin de 2003.

Les principales raisons du succès de Java sont :

* le fait qu'un programme Java peut tourner sur n'importe quel OS grâce a une machine virtuelle : la JVM .
* sa grande API : une bibliothèque standard géante (parmi les plus grandes au monde), qui nous épargne le coup de la recherche des bibliothèques externes (GUI, multi-thread, réseau... tout est déjà à votre disposition).
* son coté (presque) tout objet.

Tous ces avantages sont dus à la plateforme Java (JVM et API) et non pas au langage. On a donc créé une multitude de langages qui tournent sur la JVM pour les gens qui n'aiment pas Java (pour sa verbosité, par exemple), ou qui veulent changer un peu d'air. Parmi ces langages, on citera les plus célèbres :

* Clojure
* Groovy (++)
* Scala.

Étant donné que le seul langage statiquement typé dans cette liste est Scala, c'est lui qui a les plus grandes chances de remplacer Java, ou au moins de cohabiter avec lui, dans un futur lointain bien évidemment.

D'une part, Scala est un langage objet pur. Les classes, les constructeurs et l'héritage seront donc tous au rendez-vous mais avec une syntaxe beaucoup moins verbeuse que celle de Java, C# ou C++.

Scala supporte le paradigme fonctionnel, les fonctions sont des objets tout comme les chaines et les entiers donc elles peuvent être passées en arguments à d'autres méthodes (on parle de méthodes d'ordre supérieur dans ce cas) ou encore stockées dans des variables. Avec des bonnes fonctionnalités héritées de Haskell on peut dire adieu aux NullPointerException.

## Scala vs Java

### Préparation

On a développé dans la présentation que Scala s’exécutait sous la JVM tout comme Java. On a aussi affirmé que celui-ci était moins verbeux mais quid des performances ? Nous allons donc utiliser un algorithme simple : Tri rapide pour comparer entre les 2 langages.

La machine qui effectuera les tests est un MacBook Pro 2012 avec un processeur Intel Core i7 2640M @ 2.80 GHz et 4 Go DDR2.

Pour ce test, nous allons tester sur 100 tableaux contenant 100000 éléments un tri rapide. En 1er lieu, nous testerons de manière impérative (une seule fonction avec une récursivité) et enfin de manière fonctionnelle plus orientée objet.

### Test de scala par le code

Pour tester, il suffit simplement d’implémenter un seul package et d’exécuter nos fonctions et code dans le main(). Donc voici ce que donne le code sous SCALA :

package ***quicksort***

import java.util.Random

import scala.math.\_

object ***Main*** {

def **ARRAY\_SIZE** = 100000

/\*\*

\* Genere un tableau aleatoirement d'entiers.

\* **@return** un tableau d'entiers. La taille du tableau est defini par ARRAY\_SIZE.

\*/

def **generateArray**: Array[Int] = {

var array = new Array[Int](ARRAY\_SIZE)

val r = new Random(42)

for(i <- 0 *until* ARRAY\_SIZE) {

array(i) = r.nextInt()

}

array

}

/\*\*

\* Trie rapidement le tableau spécifié de façon impérative,

\* mais en utilisant la récursivité.

\* **@param** array tableau à trier.

\* **@param** left limite inférieure.

\* **@param** right limite supérieur.

\* **@tparam** T le type ordonnable des éléments dans la liste à trier.

\*/

def **imperativeQuickSort**[**T** <% Ordered[T]](array: Array[T], left: Int, right: Int) {

if (right > left) {

val pivot = array(right)

var p = left

for (i <- left *until* right) {

if (array(i) < pivot) {

if (p != i) {

val tmp = array(p)

array(p) = array(i)

array(i) = tmp

}

p += 1

}

}

array(right) = array(p)

array(p) = pivot

imperativeQuickSort(array, left, p - 1)

imperativeQuickSort(array, p + 1, right)

}

}

/\*\*

\* Trie rapidement le tableau spécifié de façon fonctionnelle.

\* **@param** list liste d'entier à trier.

\* **@tparam** T le type ordonnable des éléments dans la liste à trier.

\* **@return** la liste trier.

\*/

def **functionalQuickSort**[**T** <% Ordered[T]](list: List[T]): List[T] = list match {

case *Nil* => *Nil*

case head *::* tail => {

val (lesser, greater) = tail.partition(\_ < head)

functionalQuickSort(lesser) ::: head :: functionalQuickSort(greater)

}

}

def **deviation**(measures: Array[Long], mean: Long): Double = {

sqrt(measures.*map*(m => pow(m.toDouble - mean, 2)).foldLeft(0)((l,r) => l + r.toInt) / measures.length)

}

/\*\*

\* Le programme test d'abord la manière impérative et retourne le temps

\* puis test de manière fonctionnelle et fait de même.

\* **@param** args arguments du programme.

\*/

def **main**(args: Array[String]) {

val MAX\_SAMPLES = 100

var measures: Array[Long] = new Array[Long](MAX\_SAMPLES)

var total: Long = 0

for(i <- 0 *until* MAX\_SAMPLES) {

var array = generateArray

val s = *System*.currentTimeMillis()

*imperativeQuickSort*(array, 0, ARRAY\_SIZE - 1)

val e = *System*.currentTimeMillis()

measures(i) = e - s

total += measures(i)

}

var mean: Long = total / MAX\_SAMPLES

println("Tri Rapide Impératif -> temps moyen pour 100 échantillons : " + mean + "ms avec une marge de : " + deviation(measures, mean) + "ms")

total = 0

for(i <- 0 *until* MAX\_SAMPLES) {

var array = (generateArray).*toList*

val s = *System*.currentTimeMillis()

*functionalQuickSort*(array)

val e = *System*.currentTimeMillis()

measures(i) = e - s

total += measures(i)

}

mean = total / MAX\_SAMPLES

println("Tri Rapide Fonctionnel -> temps moyen pour 100 échantillons : " + mean + "ms avec une marge de : " + deviation(measures, mean) + "ms")

}

}

Vous pouvez remarquer que nous laissons une marge d’exécution puisque le programme doit d’abord générer les jeux de test. On ne mesure donc que le temps de traitement de l’algorithme.

Avec les conditions optimales de fonctionnement, nous obtenons les résultats suivants :

Tri Rapide Impératif -> temps moyen pour 100 échantillons : 39ms avec une marge de : 8.602325267042627ms

Tri Rapide Fonctionnel -> temps moyen pour 100 échantillons : 70ms avec une marge de : 15.684387141358123ms

Entre les 2 paradigmes, on constate qu’il y a une différence notoire de +40% de temps supplémentaire pour le mode fonctionnel. Il est important de souligner qu’entre écrire un code impératif et écrire un code fonctionnel, on change alors de cas d’utilisation. En paradigme fonctionnel, on recherche la maintenance pérenne ainsi que l’évolutivité de notre application. De l’autre côté, on vise la performance au détriment de la qualité du code.

Il faut aussi savoir que cet algorithme implémenté en SCALA, bénéficie d’une compilation qui permet d’optimiser des fonctions récursives en remplaçant la dernière récursion par une récursion terminale. Le principe est très simple : dans un système récursif et redondant, considère que la dernière récursion est une fonction où se trouve l’instruction finale à être évaluée. La condition d’utilisation se résume à ne pas faire de calcul ou composition. On dit alors que celle-ci doit être « pure ».

### Test de JAVA par le code

package quicksort;

import java.util.ArrayList;

import java.util.List;

import java.util.Random;

import java.util.Deque;

import java.util.ArrayDeque;

public class **QuickSort** {

public static final int *LISTE\_TAILLE* = 100000;

/\*\*

\* **Génère** **une** **liste** **aléatoirement** **d'entiers.**

\*

\* **@return** une liste d'entier. La taille de la liste est définie par LISTE\_TAILLE.

\*/

public static List<Integer> ***generateList***() {

List<Integer> aList = new ArrayList<Integer>(*LISTE\_TAILLE* );

Random r = new Random(42);

for (int i = 0; i < *LISTE\_TAILLE* ; i++) {

aList.add(r.nextInt());

}

return aList;

}

/\*\*

\* **Trie** **rapidement** **les** **éléments** **de** **la** **liste** **en** **utilisant** **une** **pile** **explicite (une** **seule** **passe)**

\* **au** **lieu** **d'un** **système** **récursif** **où** **on** **doit** **trier** **en** **plus** **chaque** **sous** **liste**

\* **qui** **sont** **à** **sortir** **de** **la** **liste** **primaire**

\*

\* **@param** aList une liste d'éléments comparables.

\* **@param** <T> type concret des éléments comparables.

\*/

public static <T extends Comparable<T>> void ***NonRecursiveQuickSort***(List<T> aList) {

Deque<Integer> stack = new ArrayDeque<Integer>();

stack.push(aList.size() - 1);

stack.push(0);

for (; stack.size() > 0;) {

int leftIdx = stack.pop(); // Borne inférieure de la sous-liste

int rightIdx = stack.pop(); // Borne supérieure de la sous-liste

if (leftIdx < rightIdx) {

T pivot = aList.get(rightIdx); // pivot

int pivotIdx = leftIdx; //index dans la liste où doit se positionner le pivot

// c'est aussi depuis cet index que la liste sera split en deux sous listes

for (int i = leftIdx; i < rightIdx; i++) {

if (pivot.compareTo(aList.get(i)) > 0) {

if (i != pivotIdx) {

T tmp = aList.get(pivotIdx);

aList.set(pivotIdx, aList.get(i));

aList.set(i, tmp);

}

pivotIdx++;

}

}

aList.set(rightIdx, aList.get(pivotIdx));

aList.set(pivotIdx, pivot);

if (pivotIdx + 1 < rightIdx) {

stack.push(rightIdx);

stack.push(pivotIdx + 1);

}

if (leftIdx < pivotIdx - 1) {

stack.push(pivotIdx - 1);

stack.push(leftIdx);

}

}

}

}

/\*\*

\* **Trie** **rapide** **sur** **une** **liste** **en** **mode** **récursif.**

\*

\* **@param** aList une liste d'éléments comparables.

\* **@param** left borne inférieur d'une sous liste.

\* **@param** right borne supérieure d'une sous liste.

\* **@param** <T> type concret des éléments comparables.

\*/

public static <T extends Comparable<T>> void ***recursiveQuickSort***(List<T> aList, int left, int right) {

if (left < right) {

T pivot = aList.get(right);

int p = left;

for (int i = left; i < right; i++) {

if (pivot.compareTo(aList.get(i)) > 0) {

if (p != i) {

T tmp = aList.get(p);

aList.set(p, aList.get(i));

aList.set(i, tmp);

}

p++;

}

}

aList.set(right, aList.get(p));

aList.set(p, pivot);

*recursiveQuickSort*(aList, left, p - 1);

*recursiveQuickSort*(aList, p + 1, right);

}

}

private static double ***deviation***(long[] measures, long mean) {

long v = 0;

for (long aMeasure: measures) {

long d = aMeasure - mean;

v += (d\*d);

}

return Math.*sqrt*(v / measures.length);

}

/\*\*

\* **Programme** **principal** **qui** **exécute** **chaque** **méthode** **l'une** **après** **l'autre.**

\*

\* **@param** args arguments console.

\*/

public static void ***main***(String[] args) {

final int MAX\_ECH = 100;

long[] measures = new long[MAX\_ECH];

long total = 0;

for (int i = 0; i < MAX\_ECH; i++) {

List<Integer> myList = *generateList*();

long s = System.*currentTimeMillis*();

*recursiveQuickSort*(myList, 0, myList.size() - 1);

long e = System.*currentTimeMillis*();

measures[i] = e - s;

total += measures[i];

}

long mean = total / MAX\_ECH;

System.*out*.println("Tri rapide récursif -> temps moyen pour 100 échantillons : " + mean + "ms avec une marge de : " + *deviation*(measures, mean) + "ms");

total = 0;

for (int i = 0; i < MAX\_ECH; i++) {

List<Integer> myList = *generateList*();

long s = System.*currentTimeMillis*();

*NonRecursiveQuickSort*(myList);

long e = System.*currentTimeMillis*();

measures[i] = e - s;

total += measures[i];

}

mean = total / MAX\_ECH;

System.*out*.println("Tri rapide non-récursif -> temps moyen pour 100 échantillons : " + mean + "ms avec une marge de : " + *deviation*(measures, mean) + "ms");

}

}

Les résultats obtenus dans des conditions optimales d’exécutions sont les suivants :

Tri rapide récursif -> temps moyen pour 100 échantillons : 13ms avec une marge de : 2.23606797749979ms

Tri rapide non-récursif -> temps moyen pour 100 échantillons : 14ms avec une marge de : 2.23606797749979ms

On remarque que l’on peut gagner 1 à 2ms avec la méthode récursive mais on peut considérer l’écart comme nulle.

### Conclusion

D’après les résultats ci-dessus, on peut déduire le graphe suivant :

Après ces tests, on peut conclure que SCALA tient ses promesses. En effet, sur l’implémentation du code, il reste beaucoup moins verbeux que JAVA et donc maintenable/évolutif. Ceci étant dit quand on va plus loin dans le code en utilisant les bonnes méthodes en JAVA (utilisation des listes d’éléments), on se rend compte que celui-ci est plus lent à terme quel que soit le paradigme.

Nous tenons à avertir que le benchmark a été effectué sur un algorithme précis. On peut comparer longtemps sur divers algorithmes de ce genre mais il faut bien se rappeler que l’on peut juger véritablement un langage sur une vraie application. Prenons l’exemple d’un site internet e-commerce : les processus métier feront essentiellement des requêtes en base de données et donc peu d’algorithme.

Dans tous les cas, nous préférons tester sur un algorithme pour déterminer plus précisément la conception verbeuse et naviguer entre différent paradigme.

## F#

Le F# fait partie lui aussi de la famille des langages à programmation fonctionnelle. Créé en 2002 par Microsoft, celui-ci est un dérivé du noyau OCaml (langage exploitant le OCaml). Ce qui lui procure l’avantage d’utiliser différent paradigme tel que l’orienté objet et l’impératif en plus du fonctionnel.

S’il y a bien une raison pour que F# devienne populaire auprès des chercheurs, c’est parce qu‘il se base sur les mathématiques.

Comme tous les langages fonctionnels, F# est basé sur les calculs lambda, qui permettent d’exprimer des concepts mathématiques facilement. Les équations peuvent être exprimées dans un format qui est similaire à leur notation mathématique.

Pour Ralf Herbrich, responsable du développement chez Microsoft Research Cambridge, il s’agit d’un aspect qui est très important, comme il l’indique dans ce témoignage: « Nous avons développé une application pour analyser des millions des feedbacks. Nous avons développé le modèle des équations sur papier et je les ai simplement tapées en tant que tel, dans un programme F# ; avec les accès à la base de données et l’enregistrement des résultats dans un fichier, le programme tenait en cent lignes de code (commentaires compris). »

Un point très fort de F# est qu’il est basé sur le Framework .NET. Il profite déjà du moteur stable et riche, avec des composants de sa bibliothèque de base qui proposent beaucoup de fonctionnalités. Comme F# peut exploiter tous les composants disponibles du Framework .NET, il bénéficie déjà d’un écosystème très riche de composants open source ou proposé par des éditeurs de logiciel. De plus, en utilisant des systèmes déjà en place dans le Framework .NET, F# est capable d’appeler du code natif en utilisant des API comme win32/win64, ou des composants COM. Mais le meilleur atout de F# est peut-être tout simplement qu’il est facile à utiliser conjointement dans des projets Visual Studio C#, pour qu’une équipe puisse exploiter le meilleur des deux langages.

En plus d’être interopérables, les programmes écrits en F# s’exécutent rapidement. Comme C#, F# est un langage compilé en un langage intermédiaire qui est ensuite recompilé, à l’exécution, en code natif, ce qui permet d’obtenir une bonne vitesse d’exécution. Nous sommes maintenant dans l’âge des processeurs multicoeurs et pour exploiter leur puissance, il faut utiliser de la programmation parallèle. Toute l’innovation dans la programmation parallèle vient de la recherche faite dans la programmation fonctionnelle et F# ne manque pas de puissance dans cette domaine. Bien sûr, des programmes F# peuvent facilement utiliser la Parallel Task Library (TPL) de Microsoft, mais en F#, il est également possible de faire de la programmation asynchrone et d’exploiter la programmation par acteur.

## F# vs C#

### Introduction préliminaire

Comme énoncé ci-dessus, F# appartient à l’environnement .NET de Microsoft. Quoi de mieux alors pour comparer les paradigmes que de prendre C# en tant que concurrent. Nous allons donc exposer les avantages et les inconvénients sur les 2 langages, exécuter un test de performance et émettre une conclusion non déterminée puisque F# et C# (bientôt M# ?) évoluent de manière considérable au fil du temps.

Pour nos tests, nous testerons en premier lieu le principe de la « curryfication » dans les 2 langages, appliquer une version simple de l’algorithme tri bulle et enfin émettre une conclusion.

La machine qui effectuera les tests est un MacBook Pro 2012 avec un processeur Intel Core i7 2640M @ 2.80 GHz et 4 Go DDR2.

### La Curryfication

Code F#

[<EntryPoint>]

do

let timer = System.Diagnostics.Stopwatch.StartNew()

let curry a b c d e f g h = a+b+c+d+e+f+g+h

let mutable i = 0

for n=1 to 10000000 do

i <- i + curry 1 2 3 4 5 6 7 8

printfn "%d iterations de i en %fs" i timer.Elapsed.TotalSeconds

System.Console.ReadLine()

Code C#

using System;

using System.Collections.Generic;

class Program

{

static void Main(string[] args)

{

var timer = System.Diagnostics.Stopwatch.StartNew();

Func<int, Func<int, Func<int, Func<int, Func<int, Func<int, Func<int, Func<int, int>>>>>>>> curry = a => b => c => d => e => f => g => h => a + b + c + d + e + f + g + h;

int i = 0;

for (int n = 1; n <= 10000000; ++n)

i += curry(1)(2)(3)(4)(5)(6)(7)(8);

Console.WriteLine(i.ToString() + " itérations de i en " + timer.Elapsed.TotalSeconds + "s");

Console.ReadLine();

}

}

« Curryfier » signifie décomposer une fonction avec de nombreux arguments en une série de fonctions qui prennent chacune un argument et produisent finalement le même résultat que la fonction d'origine. La curryfication est probablement la notion la plus délicate à saisir pour un développeur .NET, en particulier car on la confond souvent avec l'application partielle. C’est l’un des principes essentiels du paradigme fonctionnel.

L’exemple des codes ci-dessous montre l’application de cette curryfication d’une fonction sur 8 arguments différents avec une boucle d’itération de 10000000. Voici les résultats obtenus pour ces 2 codes dans les meilleures conditions possibles :

360000000 itérations de i en 1.024787s (F#)

360000000 itérations de i en 2.9634863s (C#)

Poussons le nombre d’itérations un peu plus loin pour avoir des résultats plus marqués (50000000 au lieu de 10000000) :

1800000000 itérations de i en 4.988992s (F#)

1800000000 itérations de i en 13.407076s (C#)

Les résultats parlent d’eux-mêmes, F# est 2,6 fois plus rapide que C# sur une problématique fonctionnelle. Cela s’explique par le fait que l’implémentation des delegates en .NET pour le moment inefficace et donc F# utilisent ces propres fonction de premières classes avec son propre typage : FastFunc<> contrairement à Func<> en C#.

### L’algorithme du Tri Bulles

Code F#

let inline triBulle (>) (tab: \_ []) =

for i=0 to tab.Length-2 do

for j=i+1 to tab.Length-1 do

if tab.[i] > tab.[j] then

let t = tab.[i]

tab.[i] <- tab.[j]

tab.[j] <- t

do

let tableaux = Array.init 50000 (fun i -> -float i)

let timer = System.Diagnostics.Stopwatch.StartNew()

triBulle (>) tableaux

printfn "%fs pour un tableau de %A éléments non triés " timer.Elapsed.TotalSeconds tableaux.Length

System.Console.ReadLine()

Code C#

using System;

class Program

{

static void triBulle<T>(Func<T, Func<T, bool>> greater, T[] tab)

{

for (int i = 0; i < tab.Length - 1; ++i)

for (int j = i + 1; j < tab.Length; ++j)

if (greater(tab[i])(tab[j]))

{

var t = tab[i];

tab[i] = tab[j];

tab[j] = t;

}

}

static void Main(string[] args)

{

var tabs = new double[50000];

for (int i = 0; i < tabs.Length; ++i)

tabs[i] = -i;

var timer = System.Diagnostics.Stopwatch.StartNew();

triBulle<double>(x => y => x > y, tabs);

Console.WriteLine(timer.Elapsed.TotalSeconds + "s pour un tableau " + tabs.Length + " éléments triés");

Console.ReadLine();

}

}

L’algorithme tri à bulles nous permet de nous confronter à un autre problème lors du typage des données. Comme vous pouvez le constater dans le code F#, nous avons utilisé l’instruction inline. En effet, les fonctions passées en argument dans des fonctions inline de haut niveau deviennent alors elles aussi inline et avec un type spécialisé. Ici en mettant inline, on enlève l’en-tête de l’opérateur « < » et on spécialise le type du tableau que l’on va passer en paramètre.

A partir de ces codes, nous obtenons les résultats suivants :

44.457372s pour un tableau 50000 éléments non triés (F#)

60.7896136s pour un tableau de 50000éléments non triés (C#)

Les temps ci-dessus prouvent évidement bien le concept des fonctions inline.

### Conclusion

Voici les schémas résumés des tests ci-dessus :

F# représente le côté fonctionnel de la famille .NET de Microsoft même si celui-ci reste multi paradigme. Il simplifie le code en terme de déclaratif et apporte un vrai atout sur des opérations de calcul et de pile (CPU). Cependant nous n’avons pas essayé d’élaborer des tests sur l’accès disque dur (Lecture/Ecriture fichier, Base de données). Etant donné que ce langage appartient l’univers .NET, rien n’empêche d’utiliser des librairies développées en F# pour des algorithmes complexes et d’effectuer d’autres opérations en C# par exemple.

En comparant à C#, nous nous sommes rendus compte que beaucoup de concept venaient de OCaml & Haskell et surtout un en particulier : Le pattern matching (filtrage de motif) est une fonctionnalité classique de ML/Haskell. Curieusement, on ne la retrouve que dans peu d'autres langages. Ce pattern travaille sur des valeurs et cherche à reconnaitre la forme de la valeur d'entrée. On peut voir ça comme un switch en beaucoup plus puissant. Par exemple, dans une fonction acceptant des arguments, on peut accepter un traitement sur celui-ci s’il correspond à un type particulier ou même une valeur particulière. En C#, il reste très difficile de définir un tel pattern à contrario de F# ce que se déclare très facilement :

match true with

| true -> 4

| false -> 8;;

Si vrai correspond à vrai, on retourne 4 et si vrai égal faux on retourne faux. Certains développeurs comparent cette fonction à un switch évolué car il retourne une valeur de base.

Si F# apporte de sérieux avantages dans l’optimisation et la maintenance du code, C# quant à lui, est beaucoup plus à l’aise dans les traitements et calculs sur des types en arithmétiques par exemple. De plus F# ne dispose d’une instruction tel que « goto » qui est parfois très pratique.

## ProLog

Prolog est un langage utilisé dans les domaines de l'Intelligence Artificielle et la Programmation Logique avec Contraintes. Sa syntaxe et son principe de fonctionnement sont radicalement différents de langages impératifs tels que C ou Java. Le raisonnement se rapproche plus de langages fonctionnels tels que Caml ou Lisp. Pourtant, Prolog n'est pas un langage fonctionnel. Prolog est le premier langage de programmation logique.

Le nom Prolog est un acronyme de PROgrammation en LOGique. Il a été créé par Alain Colmerauer et Philippe Roussel vers 1972. Le but était de créer un langage de programmation où seraient définies les règles logiques attendues d'une solution et de laisser le compilateur la transformer en séquence d'instructions. L'un des gains attendus était une facilité accrue de maintenance des applications, l'ajout ou la suppression de règles au cours du temps n'obligeant pas à réexaminer toutes les autres.

Comme langage de programmation, Prolog se distingue par :

* le non-déterminisme, pour la résolution de problèmes ouverts : le 'ou' utilisé en Prolog est un vrai 'ou' logique qui permet l'exploration de l'ensemble des possibles.
* la réversibilité
* la gestion des requêtes sous-contraintes/sur-contraintes

# Les performances de la technologie

Afin de tester les performances de LINQ, nous avons utilisé le projet « **LINQ Performance Test** » disponible sur le site CodeProject[[3]](#footnote-3) auquel nous avons apporté diverses modifications. Pour l’environnement Windows, nos tests ont été réalisés sur une machine DELL Latitude e6530 de 2013, équipée d’un processeur Intel Core I7 et de 8GO de RAM. Pour nos tests sous l’environnement MacOs, nous avons utilisé un MacBook Pro de 2012 équipé d’un processeur Intel Core I7 et de 4G0 de RAM.

Comme le préconise Guy Vider, auteur du projet LINQ Performance Test, nous avons utilisé un programme fonctionnant de la manière suivante :

1. Le programme alloue un *array* de n éléments et le remplie de nombres
2. Il appelle ensuite un méthode *GetAverage()* pour 3 fonctions.
3. GetAverage() appelle chacune des fonctions 1000 fois
4. Il mesure le temps qu’il faut à chaque fonction pour parcourir les n éléments et calcule une moyenne
5. Tout le programme est exécuté 5 fois pour affiner les résultats

## En environnement Windows

## En environnement MacOs

Test github

Test github2

1. WinFS était un projet de system de fichiers relationnel développé pour Windows. Le projet a été abandonné en 2006. [↑](#footnote-ref-1)
2. Dans l’environnement de programmation Microsoft, le **Common Intermediate Language** (CIL) est le langage de programmation de plus bas niveau pouvant être lu par un humain. Le code de plus haut niveau (type C#) est compilé en code *IL* qui est assemblé dans un code dit bytecode. Le code *IL* est un code assembleur orienté objet et pile qui est exécuté par une machine virtuelle. (*Source : Wikipédia)* [↑](#footnote-ref-2)
3. Le projet est disponible à l’adresse suivante : http://www.codeproject.com/Articles/21934/LINQ-Performance-Test-My-First-Visual-Studio-2008 [↑](#footnote-ref-3)