Traducteur de bytecode Java en bytecode Cil

Sylvain Pasche
EPFL

Traducteur de bytecode Java en bytecode Cil

par Sylvain Pasche

Publié 2003

Table des matières

Liste des tableaux

Liste des exemples

1. Introduction

1.1. Abstract

Certains récents langages de programmation répandus comme Java et C# utilisent la notion de machine virtuelle. Ils sont compilés vers une forme intermédiaire basée sur une définition de machine abstraite.

Pour le langage Java développé par Sun, la forme intermédiaire est le bytecode Java. Dernièrement, la plate-forme .Net développée par Microsoft utilise aussi un tel format intermédiaire connu sous le nom de "Common Intermediate Language" (abrégé Cil).

Etant donné que les deux formats intermédiaires sont relativement similaires, il serait intéressant de pouvoir passer d'un format à l'autre par le biais d'une traduction.

1.2. But du projet

Le but de ce projet est d'étudier et réaliser un traducteur de bytecode Java vers du bytecode .Net.

1.3. Cadre administratif

Ce projet est réalisé lors du 7ème semestre (hivers 2002/2003) au laboratoire des méthodes de programmation du département d'informatique de l'EPFL, sous la conduite du professeur Martin Odersky. L'assistant responsable du suivi du projet est Philippe Altherr.

1.4. Avantages et applications

L'utilisation d'une machine virtuelle et la production de code intermédiaire par un compilateur a déjà été utilisé à plusieurs reprises dans le passé. Cette façon de faire présente plusieurs avantages en comparaison à une compilation native:

- *Portabilité:* Le code intermédiaire est identique sur toutes les architectures qui interprètent ce code par une machine virtuelle. C'est l'argument principal du succès du langage Java.
- *Sécurité:* Le fait d'avoir une forme intermédiaire interprétée permet de contrôler exactement l'accès aux ressources d'un programme. Ceci permet d'exécuter du code non sûr (on peut par exemple restreindre l'accès au système de fichier pour les programmes téléchargés sur Internet). La sécurité est un élément clé des plates-formes .Net et Java. Pour cette dernière, un exemple connu est celui des des "Applets Java". Les "Applets" sont des programmes Java généralement de petite taille qui tournent dans des navigateurs Web.

- *Possibilités d'optimisation:* Le fait d'interpréter du code intermédiaire implique une baisse de performances. Les machines virtuelles récentes utilisent des compilateurs "juste à temps" (de l'anglais "Just In Time", JIT) qui traduisent dynamiquement le code intermédiaire en code natif, pour qu'il s'exécute plus rapidement. L'avantage d'avoir du code intermédiaire est qu'il est possible de bénéficier des améliorations de performance des machines virtuelles sans devoir recompiler le code source, ce qui n'est pas possible avec une compilation native.
- *Stockage de méta données (Metadata):* Le code intermédiaire ne dépendant pas d'un processeur spécifique, il est possible de stocker des données annexes dans les exécutables. Cet aspect est utilisé dans la plate-forme .Net pour stocker divers types de méta données dans le code intermédiaire, telle que le nom des champs, des données pour la sérialisation, etc.

Il est légitime de se demander à quoi peut servir d'écrire un traducteur entre les formats intermédiaires des deux plates-formes, car on peut penser qu'il serait plus utile d'écrire un compilateur Java qui émet du code Cil directement. Un tel compilateur facilite la phase de développement de code Java sur .Net. En effet, comme il a accès directement à toutes les classes, types et membres .Net, il peut détecter immédiatement les erreurs sémantiques.

Cependant, un traducteur de bytecode présente deux avantages principaux:

- Il permet d'être indépendant du langage qui produit le bytecode. Bien que la plate-forme Java a été conçue et est supportée par Sun pour ne compiler que du code Java, de nombreux autres projets produisent aussi du bytecode Java. Citons notamment le langage Pico développé par Mathias Zenger au laboratoire des méthodes de programmation de l'Epfl, ainsi que les nombreux langages référencés sur *Programming Languages for the Java Virtual Machine [VmLang]*. L'approche traducteur de bytecode permet ainsi de pouvoir traduire tout type de bytecode, indépendamment du langage qui à été utilisé pour le produire.
- Il est relativement simple à réaliser: les deux bytecodes sont basés sur un modèle de machine à pile. Ils sont donc très similaires, et une grande partie des instructions peut se traduire aisément. Cependant, la traduction n'est pas toujours évidente, et pose parfois certains problèmes complexes, comme nous le verrons plus tard.

Dans le prochain chapitre, les deux codes intermédiaires de Java et de .Net seront étudiés plus en détail, par rapport à leur historique, points communs et différences.

1.5. Structure du rapport

Le présent rapport est divisé en deux parties:

- *Partie théorique:* Cette partie détaille les procédés de traduction. Elle donne les algorithmes et les explications théoriques des différents choix. Elle tente d'être indépendante de l'implémentation, de façon à ne pas poser trop de contraintes sur les implémentations possibles.
- *Partie implémentation:* Cette partie explique dans le détail les technologies utilisées pour l'implémentation du traducteur. Elle explique les choix techniques effectués, et donne les limitations du traducteur.

2. Partie théorique

2.1. Le deux plates-formes

2.1.1. La plate-forme Java

Le langage Java est un langage de programmation orienté objet multi usage et concurrent. Sa syntaxe est similaire au C et C++. Il a été conçu par Sun Microsystem ™ et commercialisé en 1995.

La machine virtuelle Java, appelée JVM, est une machine à pile classique qui est spécifiquement conçue pour exécuter du code compilé à partir de Java. Le jeu d'instructions possède des instructions spécifiques au modèle d'objet de ce langage (héritage simple, implémentations d'interfaces, ...). La machine virtuelle travaille sur des données typées qui sont soit des types primitifs (entiers, flottants, ...) ou des références à des objets.

Pour une certaine opération, comme une addition, il existe une version de l'instruction pour chaque type applicable. Par exemple, il y a 4 instructions d'addition pour les types entier, entier long, flottant, et double flottant.

Les données peuvent se trouver soit sur la pile ou dans des variables locales. Les emplacements des variables locales et paramètres ne sont pas typés: ils peuvent contenir indifféremment n'importe quel type au cours de l'exécution, tant que les types référencés sont cohérents (contrôle effectué par le vérificateur).

2.1.2. La plate-forme .Net

Similairement à la plate-forme Java, la machine virtuelle de .Net est aussi une machine à pile. Une différence notable par rapport à Java est que les instructions ne sont pas typées, mais vont dépendre du type actuellement sur la pile. Par exemple, il y a une seule instruction d'addition pour les nombres entiers, longs où a virgule flottante.

La plate-forme est aussi basée sur le concept d'héritage simple et du système d'interfaces. La machine virtuelle, la CLR ("Common language runtime"), repose sur le principe de pouvoir exécuter plusieurs langages de programmation, ce qui fait que le jeu d'instructions est plus large que celui de Java (avec des instruction pour manipuler directement la mémoire, par exemple).

Les données de la plate-forme .Net peuvent résider dans la pile, dans les arguments ou dans des variables locales. Les emplacements des arguments et variables locales sont typés, contrairement à Java. Ceci implique qu'il n'est pas possible de stocker un entier dans une variable locale d'un autre type.

Les données peuvent être des scalaires, des références ou des instances de classes valeur ("value classes"). Les classes valeurs n'héritent pas de comportement et n'ont pas de méthodes virtuelles. Les affectations à de telles classes se font, comme le nom l'indique, par valeur. Le "Virtual object system" (VOS) permet un système de mise en boite ("boxing"): les classes valeur peuvent être stockées dans des objets alloués dynamiquement. Comme la JVM ne comporte pas d'équivalent concernant les classes valeur et le système de mise en boite, ce sujet ne sera pas traité plus en profondeur par la suite, car il n'intervient pas dans la traduction.

2.1.3. Comparaison des deux plates-formes

Les deux spécifications des machines virtuelles mettent en évidence le fait qu'elles sont basées sur un modèle similaire orienté objet. Celle de Java fait une correspondance directe entre le modèle d'objet du langage et le jeu d'instruction. Pour la CLR, le fait de pouvoir supporter des multiples langages de programmation fait que le modèle d'instruction est plus étendu.

Une autre différence majeure par rapport au design des deux machines est que les instructions de la CLR ne sont pas typées. Par exemple, l'instruction `add` de la CLR ne spécifie pas les types des opérandes dans l'instruction.

Pour plus d'informations, Gough [Gough01] fait une comparaisons des machines virtuelles JVM et CLR. Meijer et Miller [MeijerMiller01] font aussi une comparaison plutot orientée sur l'aspect multi-langages des deux plates-formes. Finalement, la thèse de Olsen [Olsen02] fournit des détails à propos de la compilation de Java sur la CLR.

2.2. Traduction des structures des deux bytecodes

2.2.1. Introduction

Cette partie décrit les méthodes et algorithmes de traduction de tout ce qui n'est pas des instructions. Cela concerne la traduction des paquetages, classes, membres, droits d'accès, etc. Pour la traduction des instructions, voir la Section 2.3.

2.2.2. Traduction des paquetages en assemblages

Unités de déploiement JVM: Au moment du déploiement, un programme Java est représenté par un ensemble de classes. Plusieures classes peuvent appartenir à un paquetage, pour former une organisation en hiérarchie. La JVM permet de spécifier des droits d'accès au niveau des paquetages, classes, et membres.

Unité de déploiement CLR: L'unité de déploiement des programmes .Net est l'assemblage ("assembly"). Les assemblages contiennent un module, qui est une représentation physique des données sur le disque. Un module est finalement composé de plusieures classes. Les assemblages permettent de stocker des informations sur les versions, des signatures cryptographiques, etc.

Problème posé: Les deux plates-formes n'utilisent pas la même notion d'unité de déploiement. La démarche pour faire une traduction entre la JVM et la CLR est donc de trouver un équivalent entre la notion de paquetages et celle d'assemblages. Il faudra donc à partir d'un nom de paquetage et d'une classe, trouver l'assemblage correspondant dans la CLR.

Algorithme de traduction: L'algorithme consiste simplement à pouvoir passer d'un couple (nom de paquetage Java, nom classe Java) à un couple (nom d'assemblage CLR, nom de la classe CLR). Il nécessite aussi des informations pour localiser les assemblages à partir des noms des paquetages Java.

Input: `package_to_assembly` table de hachage qui fait une correspondance
entre un nom de paquetage Java et un nom d'assemblage CLR.
Cette table peut par exemple contenir:

```
{ "java.lang"  ==> "vjslib"
  "java.util"   ==> "util_lib"
  "system"     ==> "mscorlib"
  "java"        ==> "vjslib" }
```

Input: `currentAssembly` le nom de l'assemblage courant de la traduction.

```
// fonction utilitaire: getParentPackage retourne le paquetage directement
//   supérieur, comme par exemple:

//   getParentPackage("org.apache.bcel") retourne "org.apache"
//   getParentPackage("java") retourne NULL



// Cette fonction permet d'obtenir le nom d'un assemblage CLR à partir
//   d'une classe Java

getCLRAssemblyName(Class : c) : string
  return getCLRAssemblyFromPackage( c.packageName )

// fonction récursive utilisée par getCLRAssemblyName
```

```
getCLRAssemblyFromPackage(string: package)
  if ( package_to_assembly.containsKey ( package )
      return package_to_assembly.get( package )
  else
      nextPackageToTry = getParentPackage( package )

      if nextPackageToTry is not NULL
         return getCLRAssemblyFromPackage( package )
      else
         return currentAssembly
```

Pour implémenter cet algorithme, il ne reste plus qu'à définir la variable `package_to_assembly`. Elle sera typiquement initialisée à partir d'un fichier de configuration.

2.2.3. Traduction des classes

2.2.3.1. Noms des classes

En ce qui concerne la traduction du nom des classes, elles vont conserver le même nom. Ainsi, la classe `java.lang.Double` se nommera aussi `java.lang.Double` une fois traduite.

Il y a cependant quelques exceptions: Java s'attend à avoir `java.lang.Object` au sommet de la hiérarchie. Par contre, la CLR aura `System.Object` au sommet de la hiérarchie des classes. Pour résoudre ce problème, il existe deux solutions:

- Substituer `java.lang.Object` par `System.Object`. Cela implique d'apporter des modifications pour que tous les membres de `java.lang.Object` puissent être hérités et accessibles dans les objects.
- Faire de `java.lang.Object` une sous classes de `System.Object`. Cette façon de faire résout les problèmes du point précédent, mais pose d'autres problèmes notamment pour la covariance des tableaux.

L'implémentation du traducteur se base sur les classes de J# de Microsoft. Comme ils ont opté pour la première solution dans leur implémentation, c'est cette première façon de faire qui sera discutée ici.

Dans le traducteur, deux cas de correspondance sont considérés:

1. java.lang.Object ==> System.Object
2. java.lang.String ==> System.String

2.2.3.2. Noms des champs

Les noms des champs peuvent se traduire directement, comme pour le nom des classes. Notons en passant que les instructions relatives aux accès des champs n'ont pas une traduction triviale. Le sujet et traité dans la Section 2.3.4.

2.2.3.3. Noms des méthodes

De manière générale, les noms des méthodes traduites comportent le même nom que les méthodes JVM. Cependant, la différence qui existe entre les hiérarchies d'objets fait que le nom de certaines méthodes peut parfois être différent. Voir la Section 2.2.3.1 pour l'explication des noms des classes.

Rennomage des méthodes héritées de System.Object. Comme `java.lang.Object` est au sommet de la hiérarchie, tous les objets héritent des méthodes définies dans cet objet. Certaines méthodes Java ont un équivalent dans la CLR qui porte un autre nom, et d'autres n'ont pas d'équivalent. Le cas des méthodes qui n'ont pas d'équivalent dans la CLR est détaillé au paragraphe suivant. Les méthodes qui ont un équivalent ne sont pas nombreuses:

Tableau 1. Equivalence entres les méthodes JVM et CLR de `java.lang.Object`

méthode JVM	équivalent CLR
`clone`	`MemberwiseClone`
`equals`	`Equals`
`finalize`	`Finalize`
`hashCode`	`GetHashCode`
`toString`	`ToString`

L'algorithme est le suivant: Si on tente d'accéder à une des méthodes JVM décrite dans la table ci-dessus (avec la bonne signature), on la renomme vers son équivalent de la CLR.

Cette façon de faire impose une restriction sur les méthodes d'une classe: une classe ne peut pas contenir simultanément une méthode JVM ci-dessus et son équivalent CLR. Par exemple, il n'est pas permis d'avoir simultanément une méthode nommée `MemberwiseClone` et `clone` (avec les signatures correspondantes). Un contrôle doit donc être effectué pour appliquer cette restriction..

Redirection des méthodes vers des classes auxiliaires. Le fait de substituer les objets JVM avec des objets CLR implique qu'il faudra rediriger certains appels de méthodes. Pour résoudre ce problème, on utilise des classes annexes, qui contiennent toutes les méthodes des objets JVM originaux, et on redirige les appels vers ces classes. Voici un exemple de ce qui se passe lors de l'appel de la méthode `trim()` sur un object de la classe `java.lang.String` qui a été substitué par `System.String`.

Exemple 1. Redirection d'une méthode vers une classe auxiliaire

Code Java:

```
"the_string".trim();
```

Bytecode JVM

```
ldc #20 <String "the_string">
invokevirtual #26 <Method java.lang.String trim()>
```

Bytecode CLR traduit

```
ldstr "the_string"
call  string [vjslib] com.ms.vjsharp.lang.StringImpl ::'trim'(string)
```

2.2.3.4. Droits d'accès

Nous avons vu comment traduire les noms des classes et membres, il reste maintenant la traduction des modificateurs. Les deux tableaux ci-dessous donnent l'équivalence entre les modificateurs des deux plates-formes. On peut noter que la sémantique des droits d'accès n'est pas toujours conservée lors de la traduction, car la notion de paquetage n'existe pas dans la CLR. Il peut donc être possible d'avoir des droits d'accès moins restrictifs dans le code traduit que le code Java original.

Accessibilité dans la CLR. L'accessibilité des membres dans la CLR est définie par deux règles: `Family` et `Assembly`. `Familiy` signifie que le membre est accessible par toutes les classes qui étendent le type. `Assembly` veut dire que le membre est accessible pour tous les types du même assemblage. Il est aussi possible d'avoir une combiaison des deux concepts: `Family and Assembly`, et `Family or Assembly`.

Accessibilité dans la JVM. La JVM ne comportant pas de notion d'assemblage, il n'y a pas d'équivalent pour l' `Assembly` de la CLR. Le détail est donné dans les tables ci-dessous:

Tableau 2. Modificateurs d'accessibilité

Entité	Modificateur JVM	Modificateur CLR
`Classes`	`public`	`public`
`Classes`	`default`	`public`
`Membres`	`public`	`public`
`Membres`	`private`	`private`
`Membres`	`default`	`assembly`
`Membres`	`protected`	`familyorassembly`

Tableau 3. Autres Modificateurs

Entité	Modificateur JVM	Modificateur CLR

Entité	Modificateur JVM	Modificateur CLR
Méthodes	synchronized	synchronized
Méthodes	native	n/a
Méthodes	final	final
Champs	volatile	n/a
Champs	transient	notserialized
Champs	final	initonly
Classes	final	sealed

2.2.3.5. Traduction des classes de base

Pour faire une traduction de programmes Java, il est nécessaire de traduire la libraire standard de Java. On pourrait penser qu'un fois le traducteur fonctionnel, il suffirait de traduire le bytecode de toute la librairie. Il se pose cependant un autre problème: la librairie comporte des méthodes natives, i.e. des méthodes qui ne n'ont pas de code JVM, mais qui existent en langage machine natif. Comme ces méthodes n'ont pas de bytecode, elles ne sont donc pas traduisibles.

Pour résoudre ce problème, il existe plusieures solutions:

- Coder en un langage .Net l'ensembles des classes natives de la libraire Java (dans un assemblage séparé, par exemple). On peut ensuite, à chaque fois que l'on rencontre une méthode native, appeler la méthode codée "à la main" de l'assemblage correspondant.
- Utiliser une libraire Java déjà portée sur .Net.

C'est la deuxième solution qui a été utilisée dans le traducteur, en utilisant la librairie J# développée par Microsoft (c'est l'assemblage `vjslib`). La limitation de cette solution est que la librairie Java de J# implémente la version 1.1.4 du JDK. Par conséquent, certaines classes et méthodes apparues dans les versions plus récentes du JDK ne sont pas disponibles.

2.2.3.6. Classe internes (Inner Classes)

Les classes internes en Java sont apparues quelques temps après la sortie du langage. Comme la notion de classes interne n'existe pas pour la JVM, les classes internes doivent être traduites en classes globales par les compilateurs Java.

Ainsi, la traduction directe des classes internes fonctionnera à l'identique sur la CLR. Cependant, vu que cette dernière possède la notion de classes interne, il serait théoriquement possible de traduire ces classes internes Java en celles de la CLR.

Dans l'implémentation, la transformation des classes internes JVM en classe internes ("nested classes") CLR n'est pas implémentée.

2.2.3.7. Traduction des types simples

La JVM comporte moins de types simples que la CLR. Cette dernière comporte des types signés et non signés, alors que la JVM ne comporte seulement des types signés, à l'exception du type `char`, non signé. La correspondance des types simples ne pose donc pas de problèmes particuliers.

Tableau 4. Traduction des types simples

Type simple JVM	Type simple CLR
`byte`	`int8`
`boolean`	`bool (int8)`
`short`	`int16`
`char`	`char (uint16)`
`int`	`int32`
`long`	`int64`
`float`	`float32`
`double`	`float64`

2.3. Traduction des instructions

2.3.1. Introduction

Beaucoup d'instructions ont un comportement similaire. Ainsi, il est possible de classifier les instructions par groupe, ce qui permet d'avoir un meilleur aperçu de la traduction.

- *Instructions de création d'objets:* Les instructions qui permettent de créer de nouveaux objets.
- *Instructions d'appel:* Les instructions pour appeler d'autres méthodes.
- *Instructions d'accès aux champs* Les instructions qui permettent d'accéder aux variables statiques ou variables d'instance des classes.
- *Instructions d'accès aux variables / paramètres:* Ce sont les instructions qui permettent d'accéder aux variables locales et aux paramètres d'une méthode. Ces accès peuvent placer une variable ou un paramètre sur la pile, ou le contraire.
- *Instructions manipulant la pile:* Ce sont les instructions qui manipulent le contenu de la pile, par empilement et dépilement.
- *Instructions de branchement:* Toutes les instructions qui peuvent avoir un effet sur le pointeur d'instruction courant. On y retrouve notamment les sauts conditionnels et inconditionnels.
- *Instructions switch:* Ce sont les deux instructions `tableswitch` et `lookupswitch`.

- *Instructions relatives aux exceptions:* Les instructions qui interviennent dans le mécanisme de gestion des exceptions.
- *Instructions à traduction triviale:* Ce sont les instructions qui ont une traduction directe, souvent avec seulement une ou deux instructions CIL correspondantes. Dans cette catégorie, on trouve les instructions arithmétiques, de conversions, d'accès aux tableaux, de comparaisons, et de concurrence.

Notation pour les instructions. De nombreuses instructions JVM ne diffère que par le type sur lequel elles s'appliquent. Pour éviter une écriture trop lourde, la convention suivante est utilisée:

`iload`, `aload`, `dload`, `fload` s'écrivent `✶load`

2.3.2. Instructions de gestion des objets

Instruction Java	Traduction IL possibles
new	newobj, et traitement spécifique

La création des objets est différente entre la JVM et la CLR. La CLR possède une instruction `newobj` qui crée une nouvelle instance d'un objet spécifié en opérande et consomme les arguments du constructeur à partir de la pile. Avec la JVM, la création d'objets se fait en deux phases:

1. L'instruction `new` empile un nouvel objet non initialisé sur la pile, la classe étant définie dans l'opérande de l'instruction.
2. Avant de pouvoir utiliser l'objet, il faut appeler la méthode constructeur avec l'instruction `invokestatic` sur l'objet non initialisé. La liste des paramètres du constructeur doit être sur la pile.

Un problème qui apparaît lors de la traduction du mécanisme de création d'un objet, est qu'il n'existe pas d'équivalent pour un objet non initialisé dans la CLR. De plus, il est possible en théorie de stocker plusieurs références à un objet non initialisé sur la pile ou dans les variables locales avant d'invoquer le constructeur. Ceci dit, un algorithme complet doit être capable d'aller rechercher toutes les références aux objets non initialisés lors de l'appel à l'instruction `invokestatic` pour initialiser toutes les instances.

Le détail d'un tel algorithme ne sera pas donné ici, car un algorithme simplifié décrit par la suite à été utilisé dans l'implémentation.

L'algorithme simplifié en question repose sur le fait que le compilateur Java produit toujours un schéma

d’instructions identique lors de la création d’objets. Un exemple de bytecode est donné ci-dessous, où l’objet `java.lang.Long` est créé avec un entier en arguement:

Exemple 2. Création d’un objet avec `new`

Code Java:

```
Long l = new Long(1);
```

Bytecode Java

```
0 new #9 <Class java.lang.Long>
3 dup
4 lconst_1
5 invokespecial #13 <Method java.lang.Long(long)>
8 astore_2
```

On peut remarquer que le `new` est suivit immédiatement de l’instruction `dup`. On peut donc traduire une telle séquence en ignorant l’instruction `new` et l’instruction `dup` et en remplaçant l’instruction `invokespecial` par l’instruction CLR `newobj`. L’algorithme consiste alors simplement à se souvenir que l’on se trouve dans une séquence d’instructions de construction d’objet pour ne pas traduire l’instruction `invokespecsial` comme dans son contexte habituel. En plus, il faudra ignorer les instructions `new` et `dup`. Pour mieux voir se qui se passe, voici le code IL traduit:

Exemple 3. Traduction des instructions `new - dup - invokestatic` pour la création d’objets

Le code Java est le même que l’exemple précédent

ByteCode CLR traduit

```
ldc.i8      1
newobj      instance void [vjslib] java.lang.Long ::’.ctor’(int64)
stloc       0
```

2.3.3. Instructions d’appel

Instruction Java	Traduction IL possibles
invokevirtual	callvirt
invokespecial	call, newobj, callvirt
invokestatic	call
invokeinterface	callvirt

La CLR possède deux instructions pour l'invocation des méthodes: `callvirt` et `call`. La JVM possède par contre 4 instructions:

- *invokevirtual* sert à invoquer les méthodes d'instance, le dispatching s'effectuant à partir de la classe.
- *invokespecial* sert à invoquer les méthodes d'instance, avec un traitement spécial pour l'appel aux super classes, méthodes privées et les méthodes d'initialisation des instances.
- *static* sert à invoquer les méthodes statiques.
- *invokeinterface* sert à invoquer les méthodes des interfaces.

En ce qui concerne `invokestatic`, `invokevirtual` et `invokeinterface`, elles peuvent se traduire directement et ne posent pas de problèmes particuliers.

Pour l'instruction `invokespecial`, il faut faire une distinction plus fine:

- Si un constructeur appelle le super constructeur il faut utiliser `call`
- Dans le cas d'une création d'une nouvelle exception, il faut utiliser `newobj`.
- Dans les autres situations, elle se traduit par `callvirt`.

2.3.4. Instructions d'accès aux champs

Instruction Java	Traduction IL possibles
getstatic	ldsfld + recherche objet destination
putstatic	stsfld + recherche objet destination
getfield	ldfld + recherche objet destination
putfield	stfld + recherche objet destination

A priori, les instructions d'accès aux champs semblent triviales, mais il y a cependant une complication mise en évidence dans l'exemple suivant:

Exemple 4. Traduction d'une instruction d'accès à un champ

Code Java

```
class A {
   int foo;
}
class B extends A {
   void m() {
      B b = new B();
      b.foo = 1;
   }
}
```

Bytecode JVM

```
 9 iconst_1
 10 putfield #14<Field B.foo int>
```

Bytecode CLR équivalent

```
IL_12:   ldc.i4.1
IL_13:   stfld  int32  A ::foo'
```

La différence est que la JVM accède à la classe `B` où le champ `foo` est hérité, alors que la CLR accède à la classe `A` où le champ est défini.

L'algorithme de traduction sera cependant assez simple: il suffit, pour chaque accès à des champs, de contrôler si le champ est bien défini dans l'objet accédé en question. Si ce n'est pas le cas, alors il faudra aller voir dans la hiérarchie de super classe jusqu'à ce que l'on trouve la classe qui contient le champ recherché.

2.3.5. Instructions d'accès aux variables / paramètres:

2.3.5.1. Instructions concernées

Instruction Java	Traduction IL possibles
✶load	ldloc✶, ldarg✶

Instruction Java	Traduction IL possibles
store	stloc✶, starg✶

2.3.5.2. Description du problème sur les deux plates-formes

2.3.5.2.1. Emplacements des variables locales et paramètres:

JVM: Sur la plate-forme JVM, les variables locales et paramètres sont stockés dans des emplacements appelés *slots*. Lorsqu'une méthode est appelée, les arguments sont stockés dans les premiers slots, et les variables locales dans les slots suivants. Le nombre de variables locales utilisés dans la méthodes est spécifiée par un paramètre du code. Les accès aux slots se font par des instructions spécifiques. Ce sont donc les mêmes instructions pour accéder à un paramètre que pour accéder à une variable locale.

CLR: Sur la CLR, il y a des instructions distinctes pour accéder aux variables locales ou aux paramètres. L'accès n'est donc pas unifié comme dans la JVM.

2.3.5.2.2. Typage des emplacements variables locales / paramètres

JVM: Sur la JVM, il est possible de stocker des éléments de n'importe quel type dans chaque slot. Il faut cependant que les types soient cohérents lorsque le vérificateur vérifie le code. Ceci dit, il est par exemple possible de stocker un entier dans le slot 0, puis de stocker un flottant quelques instructions plus tard.

CLR: Sur la CLR, les emplacements des variables locales et des arguments comportent un type, et ne peuvent seulement contenir des éléments de ce type spécifié. Il n'est pas exemple pas permis d'affecter un flottant dans un emplacement de variable locale qui comporte un type entier.

2.3.5.2.3. Allocation des emplacements variables locales / slots

JVM: Sur la JVM, certains types peuvent occuper deux slots contigus. Ce sont les types `long` et `double`. Si l'on stocke un type `long` sur le slot 4, alors les slots 4 et 5 vont contenir l'élément `long` en question.

CLR: Sur la CLR, tous les arguments ou les variables locales occupent un seul emplacement.

2.3.5.2.4. Problèmes rencontrés pour la traduction

- *Emplacements:* Un seul type d'instructions d'accès aux slots peuvent se traduire en deux familles d'instructions selon que l'on accède à un argument ou une variable locale. Une première difficulté sera donc de connaître pour chaque index de slot accédé s'il faut accéder à un argument ou une variable locale.
- *Typage:* La différence au niveau des types des variables et paramètres entre les deux plates-formes est un problème délicat: le fait de stocker deux types différents dans un même slot implique d'utiliser deux variables locales distinctes dans la CLR. Il faudra aussi faire une analyse complète du code JVM, pour connaître à quel type se réfère chaque accès à une variable locale ou paramètre.

- *Allocation:* Le fait que certains types sur la JVM occupent deux slots n'est pas un problème difficile à résoudre. Il va falloir en tenir compte lors des calculs d'index des emplacements JVM ou CLR.

2.3.5.3. Cas particuler des types référence

Gestions des types références dans les variables locales. Le fait que les slots ne sont pas typés peut poser un certain nombre de problèmes lorsque plusieurs types références différents sont stockés dans un même slot. En effet, il peut se produire deux cas de figure:

- Les types sont des variables complètement indépendantes. Dans ce cas, il faudrait au mieux utiliser deux variables locales CLR différentes, une pour chaque slot.
- L'un des type est une sous classe de l'autre. Ainsi, les deux types devraient utiliser une seule variable locale, et non deux.

Pour mieux comprendre le problème, voici un exemple simple où un même slot se voit attribuer deux types différents au cours de la méthode.

Exemple 5. Slot utilisé par deux types références

Code Java

```
{
   String s = "a string";

}
{
    Long l = new Long(1);
}
```

Code JVM

```
  0 ldc #21 <String "a string">
  2 astore_2
  3 new #23 <Class java.lang.Long>
  6 dup
  7 lconst_1
  8 invokespecial #26 <Method java.lang.Long(long)>
 11 astore_2
 12 iconst_0
 13 ireturn
```

On remarque que le type `java.lang.String` et le type `java.lang.Long` sont tous les deux stockés dans un même slot. Cela fonctionne, car la variable locale `s` n'est plus accessible dès que l'on quitte le premier bloc.

Algorithme naïf de gestion des types référence dans les slots. Comme on peut le voir dans l'exemple précédent, on pourrait se dire que l'algorithme suivant permettrait de résoudre le problème: on conserve un type référence pour chaque slot. Chaque fois que l'on stocke dans le slot un type qui n'est pas un sous type du type courant, on utilise un nouveau slot. Cet algorithme fonctionnerait bien pour l'exemple précédent, cependant il existe des cas pour lesquels l'algorithme est erroné. Voici un exemple qui ne fonctionne pas avec cet algorithme:

Exemple 6. Cas où l'algorithme naïf ne fonctionne pas

Code Java

```
class A {
  A foo() {
    A a = new B();
    a = new C();
    return a;
  }


}
class B extends A { }
class C extends A { }
```

Code JVM

```
  0 new #9 <Class B>
  3 dup
  4 invokespecial #13 <Method B()>
  7 astore_1
  8 new #15 <Class C>
 11 dup
 12 invokespecial #16 <Method C()>
 15 astore_1
 16 aload_1
 17 areturn
```

Si on applique l'algorithme précédent, on va tout d'abord affecter le type B au slot 1 à l'instruction 7. Lors de l'affectation à l'instruction 15, on voit que le type est différent et n'est pas une sous-classe. On va donc utiliser une autre variable locale. Ceci n'est pas correct, car le slot doit être partagé.

Algorithme actuel de gestion des types références . Pour résoudre ce problème, il y a deux possibilités

1. On peut faire une analyse de flux, pour connaître l'étendue de validité des variables locales, et ainsi savoir si on peut utiliser plusieurs variables pour un seul slot.
2. On utilise toujours le même slot pour tous les types référence. On place alors des instructions `castclass` si nécessaire là où il faut utiliser un type plus précis que le type déclaré.

C'est cette deuxième solution qui est utilisée ici, ainsi que dans l'implémentation, car la première se montre trop complexe à réaliser.

2.3.5.4. Algorithme de traduction

Comme mentionné plus haut, il faut récupérer l'information concernant les cadres entrants et sortants pour chaque instruction qui accède aux slots JVM. Une manière de faire est d'exécuter une vérification du bytecode comme spécifié dans la spécification de la machine virtuelle Java, tout en mémorisant les types accédés. L'algorithme en question ne sera pas détaillé ici, il peut être trouvé dans la *spécification de la machine virtuelle Java [JVM]*. Il faudra ajouter à l'algorithme une sauvegarde des cadres sortants et entrants pour chaque instruction.

La première phase de l'algorithme consiste à allouer les variables locales qui seront utilisées pendant la phase de production de code. Pour ce faire, une structure de donnée décrivant chaque slot est utilisée:

struct SlotMapper
 Map : `types` Une Map type => variable locale. Elle contient tous les
 types accédes dans ce slots
 Map : `positionToLocal` Une Map position => tuples
 (ARGUMENT / LOCALVAR, LocalVar : locvar),
 où locavar est la variable locale CLR à utiliser

Des fonctions sont utilisées pour allouer les variables locales, et faire le traitement adéquat lorsque l'on rencontre une instruction accédant à un slot.

Variables globales:

SlotMapper : `slots` un vecteur de structures SlotMapper pour
 représenter les slots Java
Types[][] : `frames` un tableau à deux dimensions calculé
 pendant la vérification du code. La première dimension
 est l'index de l'instruction courante. La deuxième dimension est
 l'index du slot ou récupérer l'information du type.

```
alloue une nouvelle variable locale:

    allocateLocalVar(Type : t) : LocalVar {
      // génère une variable locale IL du type t, et la retourne
    }

informe qu'un certain type sur un slot a été accédé:

    setAccessedType(ArgOrLocal : kind, int: slot, int: pc, Type: t)

       if types.containsReferenceType()
         oldRefType = types.getReferenceType()
         t = commonSuperType( t, types.getReferenceType() )

         types.replace ( oldRefType, t );

       end if

       if slots[slot].types.containsKey(t)
          slots[slot].positionToLocal.set( pc =>
                                          (kind, slots[slot].types(t)));
       else
          localVar = allocateLocalVar(t)
          slots[slot].types.set( t => localVar)
          slots[slot].positionToLocal.set( pos => (kind, localVar) )
       end if
```

Avec les structures de donnée et les fonctions définies, il est maintenant possible de définir l'algorithme d'analyse en question:

```
int : argIndex = 0

if method.isStatic()
  setAccessedType(ARGUMENT, argIndex++, 0, method.classType());
  foreach (argument in  method.arguments) {
     setAccessedType(ARGUMENT, argIndex++, 0, argument.Type);
     if (argument.Type == double || argument.Type == long)
       argIndex++;
}



foreach (instruction in code) {
   pc = instruction.pc
   switch (instruction)

     case ?Load (slot):
     case ?Store (slot):
        setAccessedType(LOCALVAR, slot, pc, frames[pc][slot].type);

   end switch
```

```
}
```

La deuxième partie de l'algorithme se déroule pendant la phase de production du bytecode. C'est à ce moment qu'il faudra connaître si les instructions d'accès se font sur un paramètre ou une variable locale, ainsi que la variable locale utilisée.

```
foreach (instruction in code) {
   pc = instruction.pc

   switch (instruction)

     case ?Load (slot):
     case ?Store (slot):

       switch (slots[slot].positionToLocal)
         case (ARGUMENT, localvar)
           switch (instruction)
             case ?Load (slot):
                ig.Emit(ldarg, localvar)
             case ?Store (slot):
                ig.Emit(starg, localvar)
           end switch
         case (LOCALVAR, localvar)
           switch (instruction)
             case ?Load (slot):
                ig.Emit(ldloc, localvar)
             case ?Store (slot):
                ig.Emit(stloc, localvar)
           end switch
         end switch
       end switch
}
```

Utilisation de variables locales auxiliaires. Certaines instructions nécessitent des variables locales supplémentaires pour la traduction. Par exemple l'instruction `iinc` a besoin d'une variables locale pour stocker la valeur intermédiaire que l'on a incrémenté. Un algorithme naïf serait d'allouer des variables locales lors de la traduction de l'instruction. Cette façon de faire à le désavantage de ne pas réutiliser les variables locales, et pourrait avoir comme conséquence une consommation excessive de variables.

```
fonction qui enregistre que l'on veut utiliser un certain nombre de
  variables locales à une position donnée.
  Elle est utilisée pendant la phase d'analyse

needTypes(int : position: Type[] : types) {
```

```
  ...
}

fonction qui retourne un tableau de variables locales pour une position donnée
   Elle est utilisée pendant la phase de production de code

getAuxiliaryLocals(int : position) : LocalVar[] {
  ...
}
```

Ce qu'il faut ajouter à la phase d'analyse sera donc:

```
...
case {instruction qui a besoin de variables locales}:
  needTypes(pc, [ Type1, Type2, ...]);

...
```

De même, pendant la phase de production du code, il faudra ajouter:

```
case {instruction qui a besoin de variables locales}:
  types = getAuxiliaryLocals(pc)

  ...
   // Utilisation du tableau types pour utiliser les
   // variables locales disponibles
  ...
```

2.3.6. Instructions manipulant la pile

Instruction Java	Traduction IL possibles
pop	pop
dup	dup
pop2	stloc✶, ldloc✶
dup_x1	stloc✶, ldloc✶
dup_x2	stloc✶, ldloc✶
dup2	stloc✶, ldloc✶
dup2_x1	stloc✶, ldloc✶
dup2_x2	stloc✶, ldloc✶
swap	stloc✶, ldloc✶

La JVM possède beaucoup plus d'instructions de manipulation de la pile que la CLR. Certaines instructions peuvent paraître complexes et leur utilisation peu utile à première vue, mais elles sont cependant très utiles dans certaines constructions spécifiques à Java, comme pour la manipulation des tableaux.

Pour palier l'inexistence d'une correspondance d'instructions entre les deux plates-formes, une solution possible est d'utiliser la pile. Il suffit alors d'empiler les éléments de la pile dans des variables locales, et de les dépiler ensuite sur la pile de telle manière à obtenir la même transition que celle de l'instruction JVM.

La JVM possède la notion de catégorie de type. Ces catégories représentent la taillent des éléments: 2 mots pour les types `long` et `double` et de 1 pour les autres. Cette distinction intervient dans les instructions de transition: les transitions peuvent être différentes en fonctions de la catégorie des types présents sur la pile. Voici un exemple de transition pour l'instruction `dup_x2` extrait de la *Spécification de la machine virtuelle Java [JVM]*.

Exemple 7. Transitions de pile extrait de la spécification de la machine virtuelle

> Form 1:
>
> ..., value3, value2, value1 ➠ ..., value1, value3, value2, value1
>
> where value1, value2, and value3 are all values of a category 1 computational type (§3.11.1).
>
> Form 2:
>
> ..., value2, value1 ➠ ..., value1, value2, value1

Cette distinction de transitions implique qu'il faudra calculer l'état de la pile pour l'instruction à traduire, de façon à connaître dans quel cas de la transition on se situe. On peut remarquer que les contraintes de la vérification des classes JVM nous prouvent que chaque instruction aura toujours le même type de transition. Ceci dit, il est suffisant d'avoir une traduction statique de ces instructions.

L'algorithme peut se faire en une ou deux phases. Pour éviter d'allouer inutilement des variables locales, une phase préliminaire d'analyse permet de réutiliser des variables locales allouées. L'algorithme général est le suivant:

- *Phase d'analyse*
  - *Analyse de la catégorie des éléments sur la pile* En fonction de l'instruction à traduire et de l'état calculé de la pile à l'instruction en question, on peut connaître la transition à effectuer.
  - *Allocation des variables locales pour dépiler la pile* En fonction de la transition, allouer les variables locales en bon nombre et type. Enregistrer dans une structure de donnée les opérations d'empilement et de dépilement qu'il faudra effectuer, et dans quelle variable locale il faudra stocker

l'élément de la pile. Réutiliser autant que possible d'autres variables locales allouées à des instructions de transition de la pile.

- *Phase d'émission du code*
  - Dans cette phase, il suffit de réutiliser les informations mémorisées dans la phase d'analyse pour produire les instructions de dépilement et d'empilement correspondantes.

2.3.7. Instructions de branchement

Instruction Java	Traduction IL possibles
ifeq	brfalse
ifne	brtrue
iflt	ldc.0, blt
ifge	ldc.0, bge
ifgt	ldc.0, bgt
ifle	ldc.0, ble
if_icmpeq	beg
if_icmpne	bne_un
if_icmplt	blt
if_icmpge	bge
if_icmpgt	bgt
if_icmple	ble
if_acmpeq	beq
if_acmpne	bne_un
goto	br
ifnull	brfalse
ifnonnull	brtrue
goto_w	br

Dans le cas général, les instructions de saut se traduisent facilement, car elles ont une correspondance directe dans la CLR.

Cependant, la CLR impose des restrictions quant à l'emplacement de la destination des sauts dans le contexte des exceptions. En effet, il n'est par exemple pas autorisé de sauter à l'extérieur d'un bloc `try` excepté par l'instruction `leave`. Les cas particuliers relatifs aux exceptions seront traités plus en détail dans la Section 2.3.9. L'algorithme de gestion des sauts devra donc prendre en compte si l'instruction se situe dans un bloc d'exception. Si ce n'est pas le cas, l'instruction se traduit comme indiqué dans le tableau. Si c'est le cas, il faut alors prendre des mesures spécifiques.

2.3.8. Instructions `switch`

Instruction Java	Traduction IL possibles
tableswitch	switch
lookupswitch	suites de beq et br, ou switch

2.3.8.1. les instruction `switch` et `tableswitch`

Dans la JVM, il existe deux instructions pour gérer les switch: `lookupswitch` et `tableswitch`. Dans la CLR, il n'en existe qu'une seule: `switch`.

`Tableswitch` est une instruction à longueur variable permettant d'implémenter une table de saut d'index croissants (n, n+1, n+2, ...). L'instruction CLR `switch` est très similaire à l'exception près que la table des index doit commencer à l'index 0, alors que la JVM peut spécifier l'index de départ. Pour la traduction, il suffit de soustraire un offset correspondant avant d'émettre l'instruction `switch`.

L'instruction `lookupswitch` implémente une table de sauts d'index non forcément contigus. Comme il n'y a pas d'équivalent direct pour la CLR, on peut envisager plusieurs solutions. La solution la plus simple est simplement un enchaînement de tests conditionnels et de branchements, comme dans l'exemple qui suit:

Exemple 8. Traduction de l'instruction `lookupswitch` avec enchaînement de tests

Code Java

```
switch (i) {
case 10:
case 15:
   return 123;
case 20:
case 25:
   return 124;
default:
   return 125;
}
```

Bytecode JVM

```
0 iload_1
1 lookupswitch 4: default=50
      10: 44
      15: 44
      20: 47
      25: 47
```

```
  44 bipush 123
  46 ireturn
  47 bipush 124
  49 ireturn
  50 bipush 125
  52 ireturn
```

Instructions CLR traduites:

```
    ldarg       1
    stloc       0
    ldloc       0
    ldc.i4.s    10
    beq         label_10
    ldloc       0
    ldc.i4.s    15
    beq         label_15
    ldloc       0
    ldc.i4.s    20
    beq         label_20
    ldloc       0
    ldc.i4.s    25
    beq         label_25
    br          label_default


label_10:
label_15:
    ldc.i4.s    123
    ret
label_20:
label_25:
    ldc.i4.s    124
    ret

label_default:    ldc.i4.s    125
    ret
```

Une deuxième façon de traduire une telle instruction `lookupswitch` est d'utiliser un algorithme dichotomique. L'algorithme n'est pas détaillé ici, car c'est la première façon de faire qui a été utilisée dans l'implémentation. Voici cependant un exemple pour comprendre le fonctionnement:

Exemple 9. Traduction de l’instruction `lookupswitch` avec dichotomie

Code Java

Le code Java est le même que dans l’exemple précédent.

Bytecode CLR traduit avec dichotomie

```
stloc 1 // stockage dans var auxiliaire
ldloc 1
ldc.i4 15
bge above_15

  ldloc 1
  ldc.i4 10
  bge above_10

    ldloc 1
    ldc.i4 10
    beq label_10
    br default

  above10:

    ldloc 1
    ldc.i4 15
    beq label_15
    br default

above_15:

  ldloc 1
  ldc.i4 20
  bge above_20

    ldloc 1
    ldc.i4 20
    beq label_20
    br default

  above_20:

    ldloc 1
    ldc.i4 25
    beq label_25
    br default
```

label_10:
...
label_15:
...
label_20:
...
label_25:
...
default:
...

Performance des algorithmes. L'enchaînement de tests conditionnels est l'algorithme le moins performant, car il est de l'ordre de O(n). En revanche, l'algorithme par dichotomie sera de l'ordre de O(log n). Dans le cas de switch très long exécutés régulièrement, la différence peut être notable. Cependant, la plate-forme .Net fait un usage extensif de la compilation "juste à temps" (JIT). Il est donc fort possible que la première solution soit optimisée par le compilateur, et que ses performances soient équivalentes à la solution par dichotomie.

2.3.9. Instructions relatives aux exceptions

2.3.9.1. Introduction

La traduction des exceptions de la JVM vers la CLR est un problème délicat. En effet, la CLR place beaucoup de contraintes sur les instructions et leur comportement par rapport aux blocs d'exception, alors que la JVM est assez libérale sur le sujet. De plus, les deux systèmes ont des mécanismes de fonctionnement qui diffèrent sur certains points, comme la gestion de comment parvenir vers le bloc `finally`. Pour mieux comprendre les mécanismes impliqués ici, les contraintes imposées sur les instructions des deux plates-formes seront tout d'abord discutées. Ensuite, les mécanismes qui diffèrent seront abordés, comme la gestion des instructions relatives aux blocs `finally`.

2.3.9.2. Types de blocs et gestionnaires

La CLR possède un mécanisme de gestion des exceptions plus étendu que la JVM. Elle peut contenir 4 blocs de gestion différents: `finally`, `fault`, `type-filtered` et `user-filtered` (la description exacte se trouve dans la spécification). Pour la JVM, elle possède deux types de blocs de gestion: les

blocs `catch` et `finally`. Dès lors, la correspondance logique de traduction est indiquée dans le tableau suivant:

Tableau 5. Correspondance entre les blocs d'exception JVM et CLR

bloc JVM	bloc CLR
`try`	`try`
`catch`	`catch` (`user-filtered`)
`finally`	`finally`
aucun	`filter` (`type-filtered`)

2.3.9.3. Contraintes de la CLR relatives aux exceptions

La CLR impose des restrictions beaucoup plus strictes que la JVM concernant le flot d'exécution. Les contraintes exactes sont dans la spécification de la *CLR, partition I [Ecma]*. De manière informelle, on peut les décrire comme suit: on ne peut rentrer dans un bloc que par la première instruction, et dans un gestionnaire seulement s'il y a eu une exception. On ne peut sortir d'un bloc d'exception que par une instruction `leave`.

De telles contraintes n'existent pas dans la JVM, ce qui peut poser problème pour toutes les instructions de saut. Pour résoudre ce problème, il faut analyser la destination du saut: si la destination reste dans le bloc en cours, il n'y a pas de traitement spécifique à faire. Si la destination sort du bloc, et que l'instruction est un saut conditionnel, il faudra utiliser une instruction de test avec une instruction `leave`, sinon seulement une instruction `leave`.

Un problème important dans le processus de traduction est le codage des blocs de gestion des exceptions. Dans la CLR, tous les gestionnaires d'exceptions sont spécifiés avec leurs adresses de départ ainsi que leur adresse de fin. Au contraire, la JVM n'encode pas les positions de fin des gestionnaires d'exceptions. Ceci fait qu'il est nécessaire de retrouver les positions de fin des blocs d'exception pour pouvoir les traduire vers la CLR. Cette étape n'est pas triviale et il est nécessaire de développer des algorithmes et heuristiques pour reconstituer cette information.

2.3.9.4. Algorithmes

Recherche des fins de blocs. La recherche des fins des gestionnaires ("handlers") d'exceptions de la JVM est très difficile dans le cas général, car la JVM ne pose pas de contraintes par rapport à l'emplacement des blocs de gestion des exceptions. Il serait possible, par exemple, d'avoir un gestionnaire d'exceptions qui se situe avant le bloc `try` qu'il protège. Dès lors, un algorithme gérant toutes les situations devrait faire une analyse approfondie de la structure du code.

Une simplification majeure d'une telle recherche est de partir du principe que le compilateur va placer les blocs d'exceptions dans un ordre séquentiel, avec le bloc `try` suivit d'éventuellement plusieurs blocs

`catch` et finalement d'un block `finally` facultatif. Avec cette hypothèse, on trouve 3 cas de figure, listé ci-dessous:

1. bloc try suivit de catch

   bloc try
   bloc catch
   ...
   dernier bloc catch

2. bloc try suivi de catch et finally

   bloc try
   bloc catch
   ...
   dernier bloc catch
   bloc finally

3. bloc try suivi d'un finally

   bloc try
   bloc finally

De manière générale, on peut supposer que la fin de tous les blocs non terminaux se situe juste avant l'instruction du bloc suivant.

Algorithme de recherche de la fin des blocs d'exceptions. Une étape importante des algorithmes de traduction des exceptions est de trouver la fin du bloc d'exception. Le principe de l'algorithme est de parcourir les instructions à partir d'une certaine position, afin de mémoriser la position la plus élevée de la destination des sauts. Il considère aussi une valeur minimale de destination à observer, nommée `lowestBound`. Le détail de l'algorithme est donné ci-dessous.

Input `startPc` l'offset de l'instruction du début de la recherche

Output `endPos` l'offset de l'instructions de fin de bloc trouvé

Input `lowestBound` Il faut que `endPos` > lowestBound

```
if catchBlocks.length > 0
  lowestBound = catchBlocks[catchBlocks.length - 1].start

if finallyBlock.start != NULL
  lowestBound = finallyBlock.start
```

```
pc = startPc
endPos = instructions.lastInstructionOffset
while pc < endPos; do

  switch (code[pc])
    case if*  (targetPc)
    case goto (targetPc)

      if targetPc < endPos && targetPc > lowestBound
        endPos = targetPc

  pc++
end while
```

Algorithme de recherche du début des instructions du bloc `finally` à traduire. Pour illustrer le problème, un exemple va nous permettre de mieux comprendre. L'exemple ci-dessous comporte un bloc d'exception `try-catch-finally`, avec les gestionnaires mis en évidence:

Exemple 10. Bloc `try-catch-finally`

Code Java:

```
InputStream is = null;
int i = 0;
try {
  i = is.read();
} catch (IOException e) {
  e.printStackTrace();
} finally {
  i++;
}
```

Bytecode JVM:

```
  0 aconst_null
  1 astore_1
  2 iconst_0
  3 istore_2
.finally {
.try {
  4 aload_1
  5 invokevirtual #17 <Method int read()>
  8 istore_2
  9 goto 17
} .catch java.io.IOException
```

```
 12 astore_3
 13 aload_3
 14 invokevirtual #22 <Method void printStackTrace()>
 17 jsr 31
}
 20 goto 37
.finally_handler:
 23 astore 4  <--\
 25 jsr 31    <--|  instructions à ne pas traduire.
 28 aload 4   <--|
 30 athrow    <--/
// début du bloc finally à traduire
 31 astore_3
 32 iinc 2 1
 35 ret 3
 37 return
```

Sur cet exemple, on remarque que les instructions 23 à 30 au début du bloc `finally` n'ont pas besoin d'être traduites, car elles servent au mécanisme du `jsr`. Il faut donc développer un algorithme qui nous permette de connaître le début des instructions du bloc `finally` à traduire.

Le principe de l'algorithme est de parcourir toutes les instructions à partir d'une certaine position, et de mémoriser la destination des instructions `jsr`.

Input `startPc` l'offset de l'instruction du début de la recherche
Input `lastPc` offset maximum ou effectuer la recherche

Output `finallyStart` l'offset de l'instruction ou commencer à traduire les instructions du bloc finally

```
finallyStart = -1
pc = startPc

while pc < lastPc && pc != finallyStart; do

  switch code[pc]
    case JsrInstruction (target)
       finallySTart = target
  end switch
end while

return finallyStart
```

2.3.9.5. Correspondance des exceptions lancées par la CLR

Les machines virtuelles JVM et CLR peuvent lancer des exceptions lorsque certaines conditions ne sont pas vérifiées, ou lors d'erreurs. Par exemple, une division par zéro va lancer une exception `java.lang.ArithmeticException` dans la JVM, et une exception `System.DivideByZeroException` dans la CLR. Une traduction directe des classes des gestionnaires d'exceptions ne fonctionnera pas dans tous les cas. En effet, si un programme dépend sur le fait de rattraper une exception Java de la machine virtuelle, il ne la verra pas si elle porte un autre nom lors de son lancement sur la CLR. Il faut donc implémenter un traitement supplémentaire pour ces exceptions, qui permet de fonctionner dans tous les cas.

La solution adoptée dans le compilateur J# est d'utiliser les mécanismes des filtres (gestionnaires `filter`). Si on remarque une instruction problématique lors de la traduction, il suffit d'ajouter un filtre qui va recréer une exception JVM correspondante, et va la relancer. De cette manière, le comportement sera le même que sur une plate-forme Java.

Le support des filtres n'étant pas encore implémenté dans la librairie `MSIL`, cette fonctionnalité n'est actuellement pas implémentée.

2.3.10. Instructions à traduction triviale

2.3.10.1. Instructions arithmétiques

Instruction Java	Traduction IL possibles
✶add	add
✶sub	sub
✶mul	mul
✶dev	div
✶rem	rem
✶neg	neg
✶shl	shl.un
✶shr	shr.un
✶and	and
✶or	or

Instruction Java	Traduction IL possibles
✶xor	xor

Toutes les instructions arithmétiques de la JVM ont un équivalent pour la CLR. Il n'y a donc pas de problèmes particuliers au niveau de la traduction.

2.3.10.2. Instructions de conversion

Instruction Java	Traduction IL possibles
✶2i	conv.i4
✶2l	conv.i8
✶2f	conv.r4
✶2d	conv.r8
i2b	conv.i1
i2c	conv.i1
i2s	conv.i2

De même que les instructions arithmétiques, les instructions de conversion ne posent pas de problèmes particuliers, car elles ont toutes un équivalent direct en CLR.

2.3.10.3. Accès aux tableaux

Instruction Java	Traduction IL possibles
✶aload✶	ldelem✶
✶astore✶	stelem✶

Les instructions d'accès aux tableaux sont très similaires sur les deux plates-formes. Elles nécessitent sur la pile une référence à un tableau, un index, et une valeur dans le cas d'un `store`. La valeur est ensuite stockée ou récupérée à l'index fourni.

2.3.10.4. Instructions de comparaison

Instruction Java	Traduction IL possibles
✶cmp✶	stloc, ldloc, beq, br

Les instructions de comparaison permettent de comparer deux nombres `value1` et `value2` sur la pile, et renvoient 1, 0 ou -1:

- value1 > value1 ➠ 1
- value1 == value1 ➠ 0

- value1 < value1 ➠ -1

La CLR ne comporte pas d'équivalent pour ces instructions, c'est pourquoi il faut les traduire avec plusieurs comparaisons et un saut.

> **Note :** Ces instructions sont souvent utilisées dans des tests conditionnels qui testent seulement si la valeur est nulle ou non. Une possibilité d'optimisation serait de détecter comment le résultat est consommé, pour éviter des tests et instructions inutiles.

Voici un exemple de traduction:

Exemple 11. Traduction d'une instruction `dcmpl`

Instructions JVM:

8: dcmpl

Instructions CLR traduites::

```
 stloc      3   // stockage dans la variable locale temporaire 3
 stloc      2   // stockage dans la variable locale temporaire 2
 ldloc      2
 ldloc      3
 bgt        label_emit_1
 ldloc      2
 ldloc      3
 beq        label_emit_0
 ldc.i4.m1
 br         end
label_emit_0:              ldc.i4.0
 br         end
label_emit_1:              ldc.i4.1
end:
```

2.3.10.5. Instructions de concurrences

Instruction Java	Traduction IL possibles
✶monitorenter✶	appel de `System.Threading.Monitor.Enter()`
✶monitorexit✶	appel de `System.Threading.Monitor.Exit()`

Les instructions de concurrence `monitorenter` et `monitorexit` se traduisent facilement, en appelant les méthodes statiques de la librairie de base correspondante.

3. Implémentation

3.1. Introduction

Cette partie traite plus spécifiquement de l'implémentation d'un traducteur de bytecode. Les technologies et outils existants sont passés en revue, ainsi que les choix effectués. L'implémentation est proche de la partie théorique précédente.

3.2. Parcours des technologies existantes, choix effectués

3.2.1. Introduction

Pour développer un traducteur de bytecode, il est favorable d'utiliser au maximum des librairies et outils existants pour ne pas dupliquer de travail inutilement. Ainsi, on évite de perdre trop de temps à se concentrer sur le format interne binaire d'un fichier contenant du bytecode, si on possède une bonne abstraction.

3.2.2. Librairies de lecture/écriture de bytecode Java

Java est un langage qui existe de depuis un bon nombre d'années, et est beaucoup utilisé dans des projets de recherche. Ainsi, il existe plusieurs librairies de lecture, production et instrumentation de bytecode. On peut noter par exemple *Jikes Bytecode Toolkit [JikesBT]*, *BIT: Bytecode Instrumenting Tool [Bit]*, *Gnu Bytecode [GnuBT]*, *BCEL: Byte Code Engineering [Bcel]*. On peut noter que toutes ces librairies sont écrites en Java.

BCEL est la librairie la plus utilisée dans d'autres projets, et offre le plus de fonctionnalités avancées. C'est donc la librairie choisie pour l'implémentation.

3.2.3. Librairies de lecture/écriture de bytecode CLR

La plate-forme CLR incorpore dans les classes de la librairie de base (BCL) le support pour gérer les informations du bytecode (assemblages, types, champs, ...), avec la librairie `System.Reflection`. Microsoft a rajouté une librairie similaire, `System.Reflection.Emit`dans son implémentation .Net,

permettant de générer du bytecode. L'existence de cette librairie nous offre toutes les fonctionnalités nécessaires pour générer le bytecode.

Une autre façon de produire du code CIL est de produire une version texte du code assembleur CIL et de l'assembler avec un assembleur adéquat. Un tel assembleur existe notamment sur la plate-forme .Net.

Un portage de la libraire `System.Reflection.Emit` a été écrit par Laurent Rolaz [Rolaz02] aussi dans le cadre d'un traducteur de bytecode Java vers CIL. Ce portage utilise le principe mentionné précédemment, de produire une version textuelle du code assembleur qui devra ensuite être compilé avec un assembleur. On peut cependant noter que cette librairie n'implémente qu'un petit sous-ensemble de `System.Reflection.Emit`, est qu'elle n'a pas été abondamment testée. Cela implique qu'il sera nécessaire d'investir du temps pour l'amélioration de cette librairie en cas d'utilisation pour ce projet.

Il existe cependant un problème qui se pose avec les deux outils mentionnés ci-dessus: La libraires BCEL est écrite en Java, alors que la libraire .Net pour écrire le bytecode est en C#, donc en CIL. On a alors deux possibilités pour faire face à ce problème:

1. Porter la librairie BCEL sur .Net, en la compilant par exemple avec J#. Ceci implique des modifications conséquentes sur la libraire BCEL, car J# ne possède qu'une version préliminaire de la libraire Java. Il faudrait aussi porter toutes les classes qui ne sont pas présentes dans la libraire Java de J#, mais qui sont utilisées par BCEL.
2. Utiliser le portage de la libraire `System.Reflection.Emit` en Java, de façon à pouvoir utiliser BCEL directement.

C'est la deuxième solution qui a été retenu pour le projet, pour éviter le premier problème mentionné.

Cependant, cette solution pose un problème supplémentaire: l'accès aux assemblages .Net n'est pas disponible à partir de la plate-forme Java. Il y a plusieurs solutions possibles pour résoudre ce problème:

1. Créer un fichier qui contient la description de tous les assemblages, type, et membres, par exemple sous format xml.
2. Partir du principe que tous les champs, types, et membres seront accessibles après la traduction. Un fichier de configuration est utilisé pour faire la correspondance entre les classes et les assemblages, comme mentionné dans la Section 2.

3.3. Structure du traducteur, description de l'implémentation

3.3.1. Introduction

L'implémentation du traducteur de bytecode se nomme `java2il`. Le code source, binaire, ainsi que la documentation est disponible sur "http://java2il.sourceforge.net/".

3.3.2. Comment utiliser le logiciel

La description de l'installation est détaillée dans le fichier `README.txt` qui est disponible dans l'archive du logiciel.

Utilisation. Pour obtenir un aide sur les paramètres disponibles, il suffit de passer l'option `--help` lors de l'exécution.

Exemple d'utilisation. Pour traduire un fichier `Hello.class`, par exemple.

```
./java2il --assembly Hello Hello.class
```

Etant donné que la librairie `Msil` produit un fichier sous format texte, il faut ensuite l'assembler avec un assembleur, comme par exemple celui fourni avec *"Microsoft .Net Framework"*. Exemple:

```
ilasm /out:Hello.exe Hello.exe.il
```

> **Note :** `java2il` offre la possibilité d'assembler directement les fichiers produits. Pour ce faire, il faut qu'un assembleur soit accessible par `java2il`. Il faut aussi paramétrer quel type d'assembleur utiliser, étant donné que chaque assembleur à une syntaxe différente. Le type d'assembleur devra être défini dans le fichier de configuration `java2il.properties` (plus d'informations dans la Section 3.3.5). L'option à utiliser se nomme `--assemble`.

3.3.3. Structure des fichiers Java

Description des fichiers

`Java2Il.java`

: Fichier principal qui lit les paramètres de la ligne de commande, charge les fichiers classe JVM, etc...

`AssemblyTranslator.java`

: Gère la traduction des paquetages en assemblages.

`ClassTranslator.java`

: Gère la traduction des classes.

`MethodTranslator.java`

: Gère la traduction des méthodes.

`TypeTranslator.java`

Classe utilisée pour traduire tous les types. Des méthodes statiques peuvent traduire des types simples, ou des types référence. Elle est aussi responsable de traduire les signatures des méthodes, champs, et modificateurs.

`CodeVisitor.java`

Classe abstraite pour définir des visiteurs du code des méthodes. Utilisée pour l'analyse et la production du code.

`CodeAnalysisVisitor.java`

Classe qui sert à la phase d'analyse du code. Elle va effectuer une vérification du code, pour la traduction des variables locales, comme décrit dans Section 2.3.5.

`ExceptionAnalysisVisitor.java`

Sous-classe de `CodeAnalysisVisitor`. Elle est chargée d'analyser toutes les constructions relatives aux exceptions. C'est elle qui va appliquer les algorithmes de reconstitution des portées des gestionnaires d'exceptions (voir Section 2.3.9 pour l'algorithme).

`CodeEmitVisitor.java`

Classe de génération de code. C'est ici que les instructions triviales sont directement traduites, par exemple.

`ExceptionEmitVisitor.java`

Sous-classe de `CodeEmitVisitor`, elle va définir les débuts et fins des blocs d'exception.

`MemberMapper.java`

Classe qui fait la traduction des noms des champs. C'est elle qui implémente l'algorithme de remplacement des méthodes sur les objets `System.Object` et `System.String`.

`TranslationContext.java`

Classe qui contient toutes les données globales de la traduction.

`AssembliesFactory.java`

Classe permettant de générer des références vers des assemblages. C'est dans ce fichier que l'algorithme de traduction des paquetages en assemblages est implémenté (cf. Section 2.2.2).

`LocalVariableMapper.java`

Cet objet contient toute l'information utilisée pour la traduction des accès aux variables locales et paramètres. Lors de l'analyse, des informations y sont enregistrées, et sont utilisées ensuite lors de la production du code (pour l'algorithme, voir Section 2.3.5).

3.3.4. Implémentation de la traduction des noms de paquetages

Comme expliqué dans Section 2.2.2, il est nécessaire d'avoir un fichier de configuration où stocker les paramètres relatifs à la correspondance entre les paquetages Java et assemblages .Net.

Dans le fichier `AssembliesFactory.java`, un fichier de configuration nommé `assemblies.properties` est lu lors de l'initialisation de la traduction. Voici ci-dessous un exemple d'un tel fichier:

Exemple 12. Fichier `assemblies.properties`

```
# assemblies.properties: Configuration file used to locate assemblies.
#

# assemblies signature
assembly.vjslib.ver     = 1:0:3300:0
assembly.vjslib.pubkey = B0:3F:5F:7F:11:D5:0A:3A

# package name remapping
packagemap.system         = System

# package to assemblies mapping
mapping.sun             = vjslib
mapping.org             = vjslib
mapping.java            = vjslib
mapping.javax           = vjslib
mapping.com.ms          = vjslib
mapping.System          = mscorlib

mapping.NativeCode      = classpath
```

Dans ce fichier, on trouve plusieurs informations:

- *Informations relatives à un assemblage:* Il est possible de spécifier des informations propres à un assemblage, comme la version ainsi que la clef publique. Pour ce faire, il suffit de déclarer une propriété avec un nom comme `assembly.XXX.ver` pour la version et `assembly.XXX.pubkey` pour la clef publique. Le `XXX` représente le nom de l'assemblage à paramétrer.
- *Information de correspondance paquetage => assemblage:* C'est ici que l'on peut spécifier les correspondances entres paquetages et assemblages. Pour ajouter une correspondance, il faut ajouter une propriété nommée `mapping.XXX` où `XXX` représente le nom du paquetage Java. La valeur de la propriété est l'assemblage CLR.

Le fichier de configuration `assemblages.properties` est localisé par le chargeur de classes Java. Dans la distribution standard de `java2il`, on peut par exemple modifier le fichier à partir du dossier `src/assemblies.properties`, et relancer une compilation. La compilation va copier le fichier vers le répertoire `build`, accédé lors de l'exécution de `java2il`.

3.3.5. Fichier de configuration du traducteur

Il existe un fichier de configuration `java2il.properties` qui permet de modifier certaines options de traduction, comme par exemple quels sont les objets responsables de rediriger les appels de méthode de `System.Object`.

Le fonctionnement de ce fichier de configuration est similaire à `assemblies.properties`, pour paramétrer les correspondances entre paquetages et assemblages.

Voici un exemple de ce fichier, avec un explication des paramètres.

Exemple 13. Fichier `java2il.properties`

```
skipnatives=false

# if skipnatives is not true, this is the name of the Assembly where
# to remap the Il methods
nativepackage=NativeCode

# The kind of assembler syntax supported. For now, only "microsoft"
# and "pnet" flavor are supported. For the "pnet" assembler, see
# http://www.southern-storm.com.au/portable_net.html
ilasmflavor=microsoft

# The object where to remap java.lang.Object methods
stringmapper=com.ms.vjsharp.lang.StringImpl

# The object where to remap java.lang.String methods
objectmapper=com.ms.vjsharp.lang.ObjectImpl
```

Des commentaires expliquent à quoi servent les différents paramètres. On peut voir les deux propriétés qui servent à spécifier les fichiers où rediriger les méthodes `java.lang.Object` et `java.langString`. On remarque aussi que c'est ici que l'on peut paramétrer quel assembleur utiliser, avec la propriété `ilasmflavor`.

3.4. Limitations d'implémentations

Une difficulté majeure de la traduction est sans doute la reconstitution des blocs d'exceptions. Certaines méthodes qui comportent des blocs d'exception imbriqués posent parfois problème lors de la détection des limites de ces blocs. Ainsi, le code traduit peut parfois être erroné et ne plus être vérifiable.

La gestion des exceptions de la machine virtuelle n'est pas implémentée pour le moment. Les programmes qui dépendent du fait de pouvoir rattraper ces exceptions auront un comportement indéfini lors de la traduction.

Le vérificateur de la libraire `BCEL` a été utilisé pour faire l'analyse des types. Un problème est que ce vérificateur déclare parfois que certaines méthodes ne sont pas correctes, alors qu'elles le sont (elles ont été produites par le compilateur Java de Sun). Si la vérification échoue, il n'est plus possible de continuer la traduction, car l'information sur les types n'est pas disponible. Ce problème a été rencontré lors de la traduction de la libraires Java.

3.5. Conclusions, tests

3.5.1. Tests

Des tests de traduction ont été effectués sur un petit compilateur nommé Misc développé au laboratoire des Méthodes de Programmation du Département d'informatique de l'EPFL, dans le cadre d'un cours de compilation.

Les tests sont concluants, car le compilateur généré est fonctionnel, à l'identique de la version Java.

3.5.2. Points à améliorer

Classes internes. Il serait possible d'implémenter le support des classes internes lors de la traduction, comme mentionné dans Section 2.2.3.6.

Gestion des correspondances entre exception CLR et JVM. La libraires `msil` de génération de code CIL assembleur n'implémente actuellement pas la gestion des blocs `filter`. Par conséquent, l'algorithme de correspondance entre exception CLR et JVM n'est actuellement pas implémenté. Ainsi, attraper un exception comme `java.lang.NullPointerException` ne fonctionnera pas dans les programmes traduits.

Amélioration de la recherche des blocs d'exception. Pour certaines méthodes qui contiennent des blocs d'exceptions imbriqués complexes, la traduction est parfois erronée, et le code invérifiable. Une amélioration à faire serait de mieux gérer les cas difficiles en améliorant les algorithmes.

Meilleure gestion des variables locales de type référence. Actuellement, si un slot Java peut contenir des variables locales de types différents, le type du slot est celui du plus grand super type commun de tous les types. Des instruction `castclass` sont alors utilisées. Une meilleure solution serait de faire une analyse de flux, pour connaître les régions où les variables sont utilisées. Cette analyse permettrait d'éviter des instructions `castclass` en allouant la bonne quantité de variables locales pour chaques types.

Diverses optimisations des instructions. La traduction de certaines instructions n'est actuellement pas faite de la manière la plus optimale possible. Par exemple l'instruction `tableswitch` utilise une suite de comparaisons, sans algorithme dichotomique. Un meilleur aglorithme pourrait être utilisé pour cette instruction. Un autre exemple est la traduction des instruction `*cmp*`, qui pourraient parfois être traduites plus efficacement en analysant comment sont utilisés les valeures résultat.

Extensions à d'autres implémentations de la CLR. Pour le moment, `java2il` dépend de la libraires J#, qui n'est disponible que sur Windows avec l'implémentation .Net de Microsoft. Il serait intéressant de développer l'infrastructures nécessaire pour pouvoir utiliser d'autres implémentations. Les projets d'intérêt à ce sujet sont IKVM et Mono, qui sont décrits dans la section suivante.

3.5.3. Travaux d’intérêt

Cette section décrit d’autres travaux qui sont en relation avec le traducteur de bytecode.

Travaux d’intérêt

Ikvm

IKVM.Net [Ikvm] est une machine virtuelle pour la CLR. Elle est capable de faire tourner des programmes Java sur la plate-forme CLR, elle peut aussi traduire statiquement le bytecode JVM en bytecode pour la CLR.

Ce projet est donc très similaire à `java2il`. Il est écrit en C#, et tourne sur la CLR. Il est relativement mature (environ 21k lignes de code contre 11k pour `java2il`), et est capable de traduire des programmes Java très conséquents.

Il tourne déjà sur la plate-forme .Net de Microsoft et bientôt sur l’implémentation Mono.

Mono

Mono [Mono] est une implémentation logiciel libre de la CLR. L’implémentation ne couvre pas toutes les libraires disponibles sur .Net, mais les progrès sont rapides. Un avantage de cette implémentation est qu’elle est portable sur les Unix récents, et permet ainsi d’avoir une plate-forme CLR sur GNU/Linux.

L’intérêt pour cette implémentation dans le cadre de `java2il` est qu’il serait possible, en développant le code nécessaire, d’utiliser `java2il` sur Mono. Ainsi, `java2il` ne serait plus dépendant de l’implémentation de Microsoft .Net qui le restreint à Windows.

Portable.net

Portable.Net [Pnet] est une autre implémentation de la CLR sur Unix. Elle est à la base du projet *Dotgnu* (www.dotgnu.org). Portable.Net comporte un compilateur C#, une machine virtuelle, un assembleur, un vérificateur de code et les classes de base de la CLR.

L’intérêt de ce projet pour `java2il` est qu’il est possible d’avoir un assembleur de code CIL sur Unix.

3.5.4. Pour conclure, perspectives

La traduction de bytecode entre les deux plates-formes peut sembler simple à priori, mais il existe cependant certains problèmes qui peuvent se montrer relativement complexes.

L'implémentation du traducteur à atteint un bon niveau, en permettant à des programmes de taille descente d'être traduits. L'implémentation reste ouverte aux améliorations futures.

Références

[VmLang] Robert Tolksdorf, *Vm languages (http://grunge.cs.tu-berlin.de/~tolk/vmlanguages.html)*.

[JVM] Tim Lindholm et Frank Yellin, *The Java™ Virtual Machine Specification (http://java.sun.com/docs/books/vmspec/2nd-edition/html/VMSpecTOC.doc.html)*.

[Rolaz02] Laurent Rolaz, *Traducteur de bytecodes Java en coe MSIL*, 29 juin 2002, Non publié.

[Gough01] K John Gough, *Stacking them up: a Comparison of Virtual Machines*, ACSAC-2001.

[MeijerMiller01] Erik Meijer et Jim Miller, *Technical Overview of the Common Language Runtime (or why the JVM is not my favorite execution environment)*, 8 juin 2001.

[Olsen02] Morten Sylvest Olsen, *A Pizza Compiler For .NET*, 2002.

[Ecma] Microsoft, *CLI Partition I-III (http://msdn.microsoft.com/net/ecma/)*, 2001, ECMA TG3.

[JikesBT] *Jikes Bytecode Toolkit (http://www.alphaworks.ibm.com/tech/jikesbt)*.

[Bit] *BIT: Bytecode Instrumenting Tool (http://www.cs.colorado.edu/~hanlee/BIT)*.

[GnuBT] *Gnu bytecode (http://www.gnu.org/software/kawa/api/gnu/bytecode/package-tree.html)*.

[Bcel] *The Byte Code Engineering Library (http://jakarta.apache.org/bcel/)*.

[Mono] *Le projet Mono (http://www.go-mono.org)*.

[Ikvm] *Ikvm.Net (http://radio.weblogs.com/0109845/)*.

[Pnet] *DotGNU Portable.NET (http://www.southern-storm.com.au/portable_net.html)*.