Universidade Nova de Lisboa
Faculdade de Ciências e Tecnologia
Departamento de Informática

Dissertação de Mestrado em Engenharia Informática
2º Semestre, 2013/2014

Exploração da linguagem Scala para suporte a aplicações paralelas

Nº29458 André Manuel de Oliveira Almeida Rodrigues

Orientador: Prof. Doutor Miguel Pessoa Monteiro

5 de Dezembro de 2014

Exploração da linguagem Scala para suporte a aplicações paralelas

A Deus,
sem O qual não teria acabado esta dissertação.

Aos meus irmãos,
que nos momentos difíceis me incentivaram sempre a seguir em frente.

Aos meus pais
pela ajuda, paciência e apoio.

Agradecimentos

Primeiro quero agradecer a Deus, sem O qual nunca teria acabado esta dissertação. Nos momentos mais difíceis, a ajuda veio. Pois Teu é o Reino, o Poder e a Glória para sempre. Assim seja.

Aos meus irmãos por, nos momentos difíceis e de desabafo, tiveram sempre uma palavra de apoio e conforto. O Amor esteve presente.

Aos meus pais pelo apoio e paciência, principalmente nos dias mais difíceis.

Ao professor Monteiro e ao professor Sobral, pela disponibilidade, ajuda e conselhos. Igualmente ao professor Vítor Duarte, pelas recomendações deixadas sobre como melhorar esta dissertação.

Resumo

Nos anos mais recentes tem sido feita investigação no uso da Programação Orientada a Aspectos no suporte a computação paralela, nomeadamente para conseguir guardar as funcionalidades da mesma em módulos, algo que não é possível quando se utiliza Programação Orientada a Objectos. Um resultado desta investigação foi o desenvolvimento de uma aplicação, o ParJECoLi, desenvolvida usando a linguagem Java e recorrendo ao AspectJ para paralelizar a mesma. No entanto, durante essa mesma investigação, chegou-se à conclusão que o AspectJ apresenta algumas limitações na reutilização de módulos. Tendo em conta isso, surgiu a ideia de estudar uma outra linguagem de programação, conceptualmente diferente da usada na investigação.

A linguagem Scala é conhecida por ter uma capacidade de composição modular flexível. Como tal parece interessante entender até que ponto é capaz de substituir AspectJ no suporte modular à computação paralela. Este projecto pretende aferir essa capacidade. Neste contexto, pretende-se usar o ParJECoLi como caso de estudo para dirigir comparações entre Scala e AspectJ.

Palavras-chave: Computação paralela, Scala, Modularização de facetas de paralelismo, AspectJ

Abstract

In recent years research has been done on the use of Aspect-Oriented Programming in supporting parallel computing, namely to keep its functionalities in modules, something that is not possible using Oriented-Object Programming. One result of this research was the development of an aplication, ParJECoLi, developed using the AspectJ language. However, during this same study, it was concluded that the AspectJ has some limitations in reuse of modules. Given this, the idea of studying another programming language, conceptually different from that used in the investigation emerged.

The Scala language is known to provide good support for flexible modular composition. As such it seems interesting to understand to what extent can replace the AspectJ supports modular parallel computing. This project aims to assess this ability to what extent Scala can replace AspectJ. In this context, we intend to use the ParJECoLi as a case study to be the pattern and direct comparisons between Scala and AspectJ.

Keywords: Parallel Computing, Scala, Modularization, Parallel Concern modularization, AspectJ

Índice

Lista de Figuras

Lista de Acrónimos

GAsPar:	**G**eneral-purpose **As**pect-Oriented framework for heterogeneous multicore **Par**allel systems
CP:	**C**omputação **P**aralela
SMH:	**S**istemas **M**ulticore **H**eterógeneos
POA:	**P**rogramação **O**rientada a **A**spectos
AOP:	**A**spect **O**riented **P**rogramming
POO:	**P**rogramação **O**rientada a **O**bjectos
OOP:	**O**riented **O**bject **P**rogramming
CEG:	**C**omputação **E**volucionária e **G**enética
JECoLi:	**J**ava **E**volucionary **Co**mputation **Li**brary
ParJECoLi	**Par**allel **E**volucionary **Co**mputation **Li**brary
JVM	**J**ava **V**irtual **M**achine
MVJ	**M**áquina **V**irtual **J**ava
AOSD	**A**spect-**O**riented **S**oftware **D**evelopment
SMP	**S**ymmetric **M**ultiprocessor
NUMA	**N**on-**U**niform **M**emory **A**cess
SISD	**S**ingle **I**nstruction **S**ingle **D**ata
SIMD	**S**ingle **I**nstruction **M**ultiple **D**ata
MISD	**M**ultiple **I**nstruction **S**ingle **D**ata
MIMD	**M**ultiple **I**nstruction **M**ultiple **D**ata
MPI	**M**essage **P**assage **I**nterface
PVM	**P**arallel **V**irtual **M**achine

1. Introdução

Tem havido várias investigações, nos anos recentes (Harbulot, Gurd, 2004) (Sobral, Cunha e Monteiro, 2006a) (Sobral, Cunha e Monteiro, 2006b), no sentido de melhorar a modularidade (ou modularização) de sistemas paralelos usando **P**rogramação **O**rientada a **A**spectos (**POA**, em inglês, **A**spect **O**riented **P**rogramming – **AOP**). Numa das investigações foi possível paralelizar uma aplicação, usando AspectJ, uma ferramenta que usa os princípios de POA. A aplicação usada como caso de estudo foi o **J**ava **E**volucionary **Co**mputation **Li**brary (**JECoLi**) (Pinho, Sobral e Rocha, 2012), (Pinho, Rocha e Sobral, 2010). À nova aplicação desenvolvida foi dado o nome de **ParJECoLi** (**Par**allelizing **JECoLi**) (Pinho, Sobral e Rocha, 2012).

O modelo POA surgiu do facto da **P**rogramação **O**rientada a **O**bjectos (**POO**) ser um paradigma que apresenta algumas limitações, em alguns casos, na separação e modularização de *concerns*. Define-se *concern* como uma funcionalidade ou um requisito necessário no sistema, o qual foi inserido no código do programa (Gradecki et Lesiecki, 2003). Estas limitações levam a que alguns não possam ser modularizados – os chamados *crosscutting concerns*. Existem dois sintomas que surgem devido a esta limitação (Laddad, 2012) e que definem exactamente as características de um *crosscutting concern*:

- **Dispersão do código** (*Code Scattering*) – acontece quando um *concern* é implementado ou está presente em vários módulos. Está "disperso" por esses módulos, isto é, não existe um módulo que o represente, não está confinado a um módulo.

- **Emaranhado no código** *(Code tangling)* – a dispersão de código leva ao aparecimento do segundo problema: a mistura entre o código do *concern* que é suposto o módulo implementar e do *concern* disperso. Existe assim uma mistura de código tornando-se complicado entender o módulo e o *concern* que procura implementar.

A POA propõe a utilização de uma nova unidade de modularização – aspectos – como solução para modularizar estes *crosscutting concerns* (Laddad, 2012).

1.1 Descrição do problema

Segundo (Gonçalves e Sobral, 2009), *concerns* de computação paralela entram, usualmente, na categoria de *crosscutting concerns*. Um *concern*, por exemplo, mencionado em (Sobral, Cunha e Monteiro, 2006b) é a sincronização, no acesso a recursos, o *scheduling* de tarefas, sendo que para garantir estes requisitos normalmente acaba-se por detectar dispersão e mistura de código. A adaptação de aplicações para correrem em ambientes de computação paralela leva a uma alteração intrusiva e não reversível do código fonte, o que é algo indesejável porque é prejudicial para a evolução futura do sistema, uma vez que quem programou a versão original terá sempre de ter em conta os *concerns* de paralelização (Pinho, Rocha e Sobral, 2010). Um outro motivo é porque acaba por contrariar uma das propriedades da modularização que é a capacidade de acoplar ou desacoplar módulos e, não estando um *crosscutting concern* representado ou confinado num módulo, não pode ser acoplado nem desacoplado, daí não ser reversível.

Em (Sobral, 2006), foi apresentada uma metodologia que procurou atacar o problema de *crosscuting concerns* em computação paralela, usando a POA para obter uma implementação modular desses *concerns* e, deste modo, resolver os problemas mencionados. Em (Sobral, Cunha e Monteiro, 2006a) e (Sobral, Cunha e Monteiro, 2006a), foram sugeridas colecções de aspectos, em AspectJ, para implementar padrões e mecanismos conhecidos de concorrência, demonstrando em casos ilustrativos e casos de estudo como se poderiam utilizar. Nestes trabalhos foram estudados benefícios e limitações que o uso de uma colecção poderia trazer, nomeadamente ao nível de reutilização e de desempenho.

Apesar do trabalho desenvolvido ter demonstrado que é possível resolver o problema de *crosscutting concerns*, recorrendo a AspectJ, houve, no entanto, alguns problemas na composição modular de aspectos (Sobral, Monteiro e Cunha, 2006). Um aspecto acaba por estar escondido das classes e interfaces do código fonte, assim como de outros aspectos, o que significa que não há forma de usar métodos ou informação que contenha. Esta particularidade torna, de alguma forma, difícil a reutilização de um aspecto. Um outro pormenor que dificulta a reutilização, tem a ver com o ponto referido em (Ongkingco et al, 2006) que menciona o facto de um aspecto estar fortemente acoplado a uma dada classe, faz com que viole um dos princípios da modularidade: o de esconder a sua implementação por detrás de uma interface. E a falta de interfaces torna um aspecto vulnerável, na eventualidade de existirem mudanças na classe à qual estão ligadas.

Desta forma, surgiu a necessidade de encontrar uma alternativa ao AspectJ, que pudesse emular o seu comportamento mas que oferecesse uma melhor composição modular. Deste problema, surgiu a ideia de procurar uma nova linguagem e investigar se pode apresentar-se como uma melhor solução.

O advento da linguagem de programação Scala é algo relativamente recente no mundo da programação, se tivermos em conta alguma das linguagens ainda em uso. Embora tenha começado a ser concebida em 2001, por Martin Odersky, e a primeira versão surgido em 2003, foi a partir da segunda versão desta linguagem que começou realmente a ganhar popularidade, no ano de 2006 (Scala, 2013). As principais características desta linguagem são a escalabilidade e o facto de ser multiparadigma – assenta no paradigma **orientado a objectos** e **funcional**. Um dos pontos fortes é que permite interoperabilidade com Java, ou seja, é possível utilizar o Scala para estender código de uma aplicação desenvolvida em Java sem alterar o código base. Tal é possível uma vez que o compilador daquele gera *bytecode* para correr na **M**áquina **V**irtual **J**ava (**MVJ**, em Inglês, **J**ava **V**irtual **M**achine - **JVM**). Tendo em conta estas características e outras, a serem apresentadas com mais pormenor neste relatório, seria interessante testar as capacidades desta linguagem.

1.2 Objectivos

Nesta dissertação, pretende-se por à prova a capacidade da linguagem Scala, usando o caso de estudo **ParJeCoLi**, no suporte modular à computação paralela, baseado nos princípios da POA. A implementação com Scala permitiria estudar a sua capacidade para emular o funcionamento da POA, tendo em conta que apresenta unidades de modularização diferentes, e a composição modular é um dos pontos fortes – a técnica de *mixins*, descrita em (Odersky et al, 2008) e apresentada na Secção 5.2.1 do Capítulo 5, mostra isso mesmo.

O primeiro passo deste trabalho será verificar a capacidade do Scala para emular AspectJ. No capítulo 7 são apresentadas as propostas, cada uma com características diferentes. O segundo passo será fazer um estudo comparativo entre essas propostas apresentadas, de forma a ter uma percepção de quais os seus pontos fortes e fracos, vantagens e desvantagens. Por último, será verificar se alguma dessas propostas é uma solução/alternativa face aos problemas de modularização detectados no AspectJ. Para atingir estes objectivos, será utilizado o **JEC**o**L**i como aplicação alvo e o **ParJECoLi** como caso de estudo comparativo, dado que a primeira foi a aplicação que foi estendida, na investigação da equipa da Universidade do Minho, usando o AspectJ e que resultou no desenvolvimento da segunda. Como tal, o **JEC**o**L**i serve como ponto de partida e o **ParJECoLi** como termo de comparação para verificar se o Scala tem capacidade para emular AspectJ, sendo que o **ParJECoLi** serve como referencial para efeitos de análise, comparações e tomada de conclusões. No terceiro passo, servirá igualmente para verificar se a composição modular do Scala é mais flexível do que do AspectJ. No passo dois, a ideia será usar algumas termos de comparação entre as duas propostas em Scala, (como por exemplo, número de linhas de código, número de módulos) ou verificar qual das propostas foi mais difícil e mais complexa de implementar (por exemplo, se foi mais ou menos intrusiva, levou a maiores alterações do código fonte). Põe-se a hipótese de usar métricas, para medir desempenho por exemplo,

embora isso só se possa verificar quais são credíveis de utilizar depois de implementar as propostas.

1.3 Principais Contribuições

As principais contribuições resultantes da elaboração desta dissertação serão:

- Aferir a capacidade do Scala para emular os princípios do AspectJ tendo um modelo conceptual diferente das linguagens existentes;
- Aferir a capacidade do Scala no suporte à computação modular, baseado nos princípios da POA;
- Apresentar estudo comparativo das soluções (ou propostas) em Scala para emular AspectJ, verificando quais os seus pontos fortes e fracos;
- Verificar, dentro das soluções apresentadas, se a composição modular é uma alternativa aos problemas detectados no AspectJ;
- Permite aferir limitações de interoperabilidade entre Scala e Java;

1.4 Organização do documento

Para além do presente capítulo, a estrutura desta dissertação assenta em mais quatro capítulos, organizados pela seguinte ordem:

- Capítulo 2 – apresenta uma descrição da Programação Orientada a Aspectos: o que motivou a criação deste modelo de programação e quais os seus princípios. É apresentado igualmente a linguagem AspectJ e quais os seus mecanismos;
- Capítulo 3 – é introduzido a área de Computação Evolucionária e Genética, dado que este é o domínio da aplicação alvo JECoLi, sendo por isso necessário ter algum domínio sobre os conceitos que trata;
- Capítulo 4 – é um capítulo que faz uma apresentação não muito profunda, sobre a computação paralela;

- Capítulo 5 – é dedicado à linguagem Scala. É feita uma introdução desta linguagem e os mecanismos que se prevê usar neste trabalho;

- Capítulo 6 – é referenciado o trabalho de pesquisa, relacionado com o tema;

- Capítulo 7 – são apresentadas as propostas pesquisadas e o seu uso para emular o funcionamento de Aspecto. É feita uma descrição do trabalho realizado, nomeadamente sobre a implementação dos aspectos do ParJECoLi, com as soluções encontradas;

- Capítulo 8 – são apresentadas as conclusões, tendo em conta os objectivos propostos, e sugestões para trabalho futuro;

Este trabalho está inserido no contexto do projecto GAsPar (General-purpose Aspect-Oriented framework for heterogeneous multicore Parallel systems), financiado pela FCT (Fundação para a Ciência e Tecnologias), com contribuições na área de **C**omputação **P**aralela (**CP**) (Lin e Snyder, 2009), utilizando princípios de **P**rogramação **O**rientada a **A**spectos (**POA** ou, em inglês, **A**spect **O**riented **P**rogramming – **AOP**) (Laddad, 2012). O seu objectivo é desenvolver novas técnicas e ferramentas que solucionem alguns dos problemas encontrados dentro da área de CP.

2. Programação Orientada a Aspectos

A POO é actualmente o paradigma de programação dominante (Meyer, 1997). Um grande ponto forte deste paradigma é o facto de poder representar um *concern* como um módulo, e assim sendo uma instância sua possui estado – devido aos seus membros, que são variáveis ou campos internos - e que cada objecto pode apresentar um conjunto de métodos que podem retornar dados sobre o objecto e/ou alterar o estado do mesmo – dos seus membros. Para além disso, apresenta outros pontos positivos como a abstracção de classes, esconder informação, substituição modular – poder substituir um módulo por outro sem impacto para os restantes - e capacidade de acoplar e desacoplar módulos (Monteiro e Fernandes, 2011).

No entanto este paradigma apresenta algumas limitações no que diz respeito à separação de *concerns*: o facto de usar um único critério de decomposição leva a que alguns *concerns* não possam ser modularizados – os *crosscutting concerns*. Existem dois sintomas que surgem devido a esta limitação (Laddad, 2012), já referidos, a dispersão do código e o emaranhado no código.

Todos estes problemas levam a outros problemas como rastreabilidade (*poor traceability)*, fraca reutilização de código, baixa produtividade, baixa qualidade e evolução difícil (Laddad, 2012).

Dados todos estes problemas, foi necessário encontrar uma solução que resolvesse as questões que a **POO** não conseguia resolver por si só. Foi dessa necessidade que surgiu a **P**rogramação **O**rientada a **A**spectos (**POA,** em inglês **A**spect **O**riented **P**rogramming - **AOP**). A **AOP** é um modelo de programação que permite a separação de *crosscutting concerns* (em português foi traduzido como facetas transversais), ao introduzir uma nova unidade de modularização chamada *aspect* (Laddad, 2012). Cada *aspect* pretende modularizar uma funcionalidade transversal. A **POA** surgiu devido a algumas das limitações que a **P**rogramação **O**rientada a **O**bjectos (**POO**) apresenta, nomeadamente ao facto de não conseguir manter a separação dos *crosscutting concerns* de um sistema. Mais à frente este assunto será mais aprofundado.

Para melhor entender **AOP**, torna-se necessário entender o que são *concerns* e o que são *aspects*. Como foi dito anteriormente, um *concern* é uma "preocupação", que retracta uma funcionalidade do sistema que se vai implementar. Cabe ao engenheiro de software estudar e compreender qual é a melhor forma de implementar esse *concern* dentro do sistema.

Podemos considerar dois tipos de *concerns*: *core concern* e *crosscutting concern*. A classificação de um ou outro *concern*, num sistema a implementar, depende do paradigma de programação a ser utilizado. Os *core concerns* são, tal como o nome indica, funcionalidades base de um dado sistema. Por exemplo, num sistema de *banking* implementado com o paradigma de programação orientado a objectos (**POO**), considerar-se-ia como funcionalidades base a gestão de contas, gestão de clientes e a gestão do fundo. São *concerns* que são implementados por um módulo e não vão estar presentes em mais nenhum outro. Se considerarmos, dentro do mesmo sistema, o requisito autenticação, podemos estar perante um *crosscutting concern*. Por exemplo, se cada vez que um utilizador do banco ou gestor de conta quiser aceder a dados, antes de o fazer tiver de se autenticar, significa que em vários módulos vai estar presente código fonte que verifica se o utilizador já está autenticado ou não (Laddad, 2012). Dado que este *concern* vai estar presente em vários módulos, significa que essa faceta (*concern*) é transversal aos módulos em questão.

Um *aspect* é uma nova unidade de modularização introduzida pela **AOP** cujo objectivo é implementar a funcionalidade *crosscutting concern*. A utilização de *aspects* permite uma melhor separação de *concerns*. Pode ter membros privados, pode ter métodos e da mesma forma é possível inicializar como um objecto. No entanto, as suas funcionalidades não funcionam da mesma forma das de uma classe. Para correr um *aspect* não é necessário instanciá-lo e inicializá-lo como um objecto e, da mesma forma, as suas funcionalidades - *pointcut* e *advice* - não podem ser chamadas através do objecto declarado.

Dito isto, o que é que diferencia a AOP de outros paradigmas de programação? Segundo (Filman & Friedman, 2000) existem duas propriedades que a distinguem de outros paradigmas: *quantification* e *obliviousness.*

- *Quantification* é a capacidade de executar um pedaço de código em vários pontos diferentes de execução de um programa. No fundo algo como "sempre que esta condição se verificar, executa este pedaço de código". Se considerarmos uma implementação da AOP – a linguagem AspectJ – vemos que a propriedade *quantification* é conseguida através de um conceito chamado *joinpoint.* A abordagem baseada em *joinpoints* permite executar uma acção num determinado ponto da execução do programa. Na secção de AspectJ este ponto é aprofundado com mais pormenor.

- *Obliviousness* é a capacidade de quantificar um pedaço de código num programa onde não houve a preocupação da parte dos programadores que esse código pudesse ser acrescentado futuramente. Ou seja, é possível, usando o paradigma AOP para acrescentar código em programas que não foram desenhados ou criados com a preocupação de que pudesse ser acrescentado *aspect modules*.

Em jeito de conclusão podemos dizer que a AOP é comparável à folha de estilos CSS. Da mesma forma como o CSS age sobre o html, assim também os *aspects* agem sobre as classes (Laddad, 2012).

2.1 AspectJ

O AspectJ é uma linguagem de programação que se baseia no paradigma AOP. Existem muitas outras linguagens que também implementam este paradigma mas o AspectJ é a linguagem mais conhecida e robusta. Esta linguagem estende a linguagem Java, orientada a objectos, para poder implementar um novo módulo - *aspects*.

A Figura 2. 1 mostra a implementação de um *aspect* em Java:

```
package hello;

public aspect Answer1a {
    pointcut callsToGreeting(): call(public void HelloWorld.greeting());

    after(): callsToGreeting()  {
        System.out.println("After Hello World");
    }

}
```

Figura 2. 1 - Exemplo de um Aspect usando AspectJ.

Para além do *aspect*, a funcionalidade do AspectJ assenta também em mais dois mecanismos: o *pointcut* e o *advice*.

Para entender como funciona os *pointcuts* e os *advices*, primeiro é necessário explicar o conceito de *joinpoint* (ponto de junção). Um *joinpoint* é um ponto no código fonte onde queremos aplicar um *advice* definido num aspecto, permitindo assim acrescentar ou modificar o comportamento da aplicação alvo sem alterar o código base. Um ponto de junção, na maioria das ferramentas existentes, é seleccionado nomeando membros de uma classe em que se pretende alterar o comportamento. Podem ser métodos – um método é membro de uma classe- ou acesso a campos. Um *joinpoint* é essencialmente dinâmico, i.e., é um evento que ocorre durante a execução do programa. Uma das consequências de utilizar este tipo de pontos de junção é que diminui a possibilidade de reutilização de aspectos, dado que estes estão fortemente acoplados à classe alvo (Monteiro e Fernandes, 2011).

Um *pointcut* é a expressão que identifica os pontos de junção que se pretende alcançar. Pode-se referir a um método ou um objecto. Pode-se dizer que, no fundo, é o mecanismo que especifica um ponto de junção entre o código base e o aspecto base. Um *pointcut* pode ser anónimo - e como tal está embebido juntamente com um *advice* ver Figura 2. 2 – ou pode ter

um nome e, como tal, pode ser chamado ou referenciado por um *advice* – ver Figura 2. 1. Este último é mais prático, uma vez que permite que um *pointcut* possa ser utilizado por mais do que um *advice*.

```
public aspect Answer1e {
   void around(): call(public void HelloWorld.greeting()) {
      System.out.print("[");
      proceed();
      System.out.println("]");
   }
}
```

Figura 2. 2 – Definição de um *advice* usando *pointcut* anónimo.

Pointcuts são constituídos por construções elementares chamados *pointcut designators* – também referidos como *pointcuts* de segundo nível – que no fundo permitem especificar em que contexto se pretende capturar aquele ponto de junção. Estes mecanismos são:

- *Within* – limita a acção de captura a uma classe ou *package*.
- *Get* - captura todos os acessos a um dado objecto.
- *This* - captura o objecto que tem o controlo naquele ponto de execução.
- *Call* – designador que permite capturar a chamada a um método quando este é invocado no código base.
- *Execution* – designador que permite capturar a chama a um método mas cujo o momento de captura é diferente do *call*.

2.1.1 Call vs Execution

A diferença entre *call* e *execution* consiste no momento em que a chamada ao método é interceptada: no primeiro caso, o método é interceptado antes do controlo passar do objecto invocador para o objecto invocado, no segundo é o contrário (AspectJ, 2014) e (AspectJb, 2014). Se utilizarmos o PCD *this* no primeiro caso, o objecto que é retornado é aquele que invoca o método, enquanto no segundo caso é o objecto invocado, ao qual pertence o método. Isto acontece porque durante o fluxo de execução do programa, um objecto que tem o controlo no momento e se invoca um método de um outro objecto, o controlo é transferido para o objecto invocado até que o método seja executado, sendo que no momento a seguir retorna de volta para o objecto invocador. Daí que os objectos capturados sejam diferentes, porque o momento de captura do método é diferente.

Figura 2. 3 - Mecanismos do AspectJ no exemplo anterior

O *advice* (ver Figura 2. 3) é o mecanismo no AspectJ que vai agir sobre o ponto de junção. Para um *advice* ser utilizado, precisa especificar o *pointcut*, para obter o ponto de junção no código base sobre o qual vai agir. Existem três tipos de *advice*:

- Before - permite ao aspecto executar uma acção antes do código ser executado.
- After - permite ao aspecto executar uma acção depois do código ser executado.
- Around – mecanismo que permite contornar a execução do código, ou seja, o código capturado pelo *pointcut* não é executado. Para forçar a execução do código, pode-se usar a função *proceed()*. Usando esta função mais do que uma vez, leva a que o código capturado seja executado mais uma vez.

2.1.2 Weaving

Dito de uma forma simples, o processo de fazer *weaving* (traduzido para português, tecelagem) consiste em juntar o código dos aspectos com o código fonte. O AspectJ utiliza o seu próprio compilador, diferente do compilador Java, mas que produz o mesmo tipo de código – *bytecodes* - que corre na máquina virtual Java. Acontece que o compilador corre os aspectos e, consoante o que ler, vai juntar, vai tecer as partes do aspecto ao código fonte, mais precisamente à classe ou interface, criando novas versões desta com o código do aspecto (ver Figura 2. 4) (AspectJ, 2014).

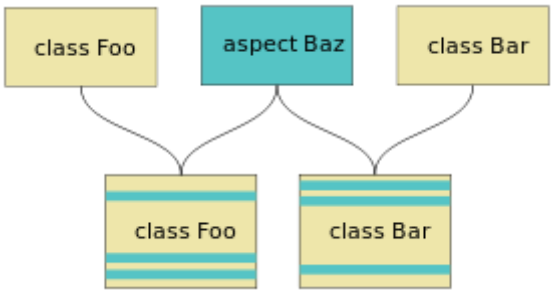

Figura 2. 4 – Ilustração do processo de *weaving* entre classes e um aspecto

O processo de *weaving* pode acontecer de 3 formas diferentes:

- **Compile-time weaving** – é a forma mais simples do processo ocorrer: o compilador tem acesso aos ficheiros do código fonte, tem acesso aos aspectos e gera ficheiros binários que são uma junção do código fonte com partes dos aspectos respectivas (ver Figura 2. 4 – primeiro nível código fonte e aspecto, segundo nível ficheiros binários tecidos);
- **Post-compile weaving** (ou **binary weaving**) – consiste em injectar instruções de aspectos em classes já compiladas;
- **Load-time weaving** – consiste em carregar classes tecidas, ou seja, compiladas com partes de aspectos, em vez de carregar as classes originais ou limpas, isto é, sem influência de aspectos. É interessante para casos em que o tipo de execução de uma aplicação depende das escolhas do utilizador, e estas podem resultar em usar ou não essas classes já tecidas com código de aspecto(s).

2.1.3 Declaração de Inter-Tipos

Declaração Inter-Tipo (em inglês Inter-Type Declaration, as vezes referido por ITD) é referida em (Laddad, 2012) como uma construção de ou para facetas estáticas (por oposição aos *advices*, construções de ou para facetas dinâmicas). A diferença é que facetas dinâmicas adicionam e até modificam o comportamento do código fonte – através dos *advices* - mas não alteram a sua estrutura. Se quisermos alterar a estrutura de uma classe ou interface, seja adicionando um novo membro privado ou um novo método, então queremos uma construção que altere a estrutura de uma classe de modo a implementar uma faceta, que neste caso é estática – ou antes, queremos implementar a parte estática de uma faceta alterando a estrutura do código (Gradecki et Lesiecki, 2003).

Algumas alterações que o uso de declarações intertipo permite (Gradecki et Lesiecki, 2003):

- Adicionar membros (membros, métodos, construtores) a tipos (incluindo outros aspectos);

- Adicionar uma implementação concreta a interfaces;
- Declarar que tipos estendem novos tipos ou implementam novas interfaces;
- Declarar precedência de aspectos;

Embaixo podemos ver um exemplo de um aspecto que adiciona membros e métodos à interface IAlgorithm:

```
public aspect DistributedMigration extends AbstractMigration{

	private boolean syncronized;
	Graph gr;
	ArrayList<ISolution[]> indivs=new ArrayList<ISolution[]>();
	private ArrayList<Integer> aux=new ArrayList<Integer>();
	private static ExecutorService pool= Executors.newCachedThreadPool();

	(…)

	public int IAlgorithm.IterationStep;

	public int IAlgorithm.migrants;

	public boolean IAlgorithm.synchro;

	public void IAlgorithm.setIterationStep(int it)
	{
		IterationStep=it;
	}

	public int IAlgorithm.getIterationStep()
	{
		return IterationStep;
	}
}
```

Existem vários pormenores a ter em conta quando se utilizam inter-tipos. Um deles é que se alterarmos a estrutura de uma interface, automaticamente todas as classes que a implementam, quando instanciados, vão ter acesso a esses membros e métodos, a não ser que estes sejam declarados como privados, o que significa que apenas o código dentro do aspecto tem acesso aos mesmos (significa que o aspecto vai ter acesso aos métodos e membros adicionados ao objecto, pelos inter-tipos, quando este for capturado por algum dos *pointcuts*). A visibilidade *protected* não existe em aspectos. Um outro pormenor é que o aspecto pode adicionar membros a uma interface (o que acaba por ser um contra-senso, isto até ao Java 8, onde nunca existiu o conceito de *traits*, um tipo de interfaces com estado – ver secção 5.2.1 para mais informação).

Se tivermos duas instâncias de classes com membros que foram adicionados por um aspecto, vamos imaginar um contador, por exemplo, o estado dessa variável pode ser diferente nas duas, depende das operações que se realizarem sobre elas. Isto para dizer que os ITD´s revelam-se ao nível da instância de um objecto, ou seja, estendem, alteram a estrutura

de uma classe ou de qualquer outro módulo dentro do Java, tal e qual como se essa alteração fosse dentro da própria classe. O processo de *weaving* entre aspectos e classes ou interfaces é mais evidente quando se fala de ITD´s, dado que podemos ver que as extensões que um aspecto produz são "tecidas", são unidas.

2.2 AspectJ em ParJeCoLi

O Anexo A mostra duas tabelas com informação que pode ser considerado como um levantamento de mecanismos do AspectJ presentes no ParJECoLi. A Tabela A. 1 é um levantamento de *pointcuts* com nome, ou seja, que podem ser referenciados por *advices*. A Tabela A. 2 é um levantamento de todos os *advices* existentes, assim como detalhes sobre o seu *pointcut* que utiliza – se for *pointcut* com nome, está indicado qual é e basta apenas consultar a Tabela A. 1 para saber os seus detalhes. Se for *pointcut* anónimo, estão presentes os *pointcut designators* que utiliza. Os *advices* estão organizados por tipo (*before*, *after* e *around*) e é indicado em que aspecto está implementado.

A intenção deste levantamento foi para permitir, em primeiro lugar, aprofundar o conhecimento sobre AspectJ e a forma como foi utilizado no JECoLi para estender a aplicação. Em segundo lugar, porque facilitou a implementação das propostas no capítulo 7, dado que permitiu consultar de forma rápida os detalhes dos aspectos existentes sem ter de os procurar no código fonte da aplicação.

3. Computação Evolucionária e Genética

O campo da **C**omputação **E**volucionária e **G**enética (**CEG**) é um ramo da informática que surgiu há sensivelmente três décadas e que emergiu com a necessidade de encontrar uma solução para os problemas de optimização de procura que começaram a surgir. A inspiração para os algoritmos sobreveio da observação à forma como a natureza buscava resolver os seus próprios problemas de optimização - selecção natural, só os mais aptos ao ambiente ou às condições ou restrições que os rodeiam é que sobrevivem. Esta linha de pensamento levou ao desenvolvimento de métodos como os **A**lgoritmos **E**volucionários (**AE**), **A**lgoritmos **G**enéticos (**AG**), **E**stratégias de **E**volução (**EE**), **P**rogramação **E**volucionária (**PE**) ou **P**rogramação **G**enética (**PG**). Até aos dias de hoje, estes métodos ganharam especial relevância pela capacidade de resolver um vasto leque de problemas, tanto no campo científico como no campo tecnológico, com grande sucesso. Da mesma forma foram surgindo, ao longo dos tempos, variantes destes algoritmos sendo talvez os mais conhecidos, os **A**lgoritmos **M**eméticos (**AM**) ou a **E**volução **D**iferencial (**ED**) (Pedro Evangelista et al).

O campo da **CEG** ganhou tal relevância dentro da Informática que hoje em dia existem vários livros, conferências e publicações exclusivamente dedicados ao assunto. Tal crescimento levou, naturalmente, à criação de várias ferramentas, plataformas *open-source*, que providenciam implementações destes métodos, disponíveis para qualquer utilizador ou investigador os pudesse utilizar, aplicando a ferramenta a um problema específico (Pedro Evangelista et al).

Neste trabalho vai ser dado mais ênfase aos **AE**, em especial à sua paralelização, uma vez que a aplicação alvo – a *framework* **JECoLi** – implementa alguns destes algoritmos. Para experimentar a aplicação e compreender os resultados, depois de paralelizar a sua execução com a biblioteca a desenvolver, torna-se importante compreender o funcionamento dos algoritmos.

Os **AE** são métodos de procura estocásticos que foram usados para a resolução de problemas de procura, optimização e para o problema de aprendizagem-máquina (Alba e Tomassini, 2002). O modo de funcionamento padrão, deste tipo (ou classe) de algoritmo, é o seguinte: por norma define-se um número finito de iterações, em que em cada iteração será gerada uma nova população a partir de uma população antiga. Uma população tem um conjunto de indivíduos que representam uma versão codificada (binário, real, inteiro,…) de uma possível solução. Uma função de avaliação atribui um valor, chamado *fitness*, que no

fundo é uma medida da aptidão do indivíduo para ser a solução do problema (de optimização, de procura, …). Na versão geral do algoritmo, são aplicados operadores de variação para obter uma população temporária, avaliá-la (usando a função de avaliação) e a seguir obter uma nova população usando a população original do início da iteração ou a população temporária gerada. Este algoritmo está representado na fig. 3.1. Na versão canónica deste algoritmo, normalmente é aplicado um operador estocástico – selecção, mutação, *crossover*, recombinação – logo no início numa população aleatória, de forma a obter uma nova população, com novos indivíduos (Alba e Tomassini, 2002).

```
Evolutionary Algorithm
t := 0;
initialize and evaluate [P(t)];
while not stop_condition do
  P'(t) := variation [P(t)];
  evaluate [P'(t)];
  P(t+1) := select [P'(t),P(t)];
  t := t + 1;
end while;
```

Figura 3.1 – Modelo padrão de um Algoritmo Evolucionário (Alba e Tomassini, 2002).

3.1 Algoritmos Panmícticos

Um **AE** panmíctico baseia-se num modelo no qual toda a população é lidada como se fosse apenas um único conjunto de indivíduos e, como tal, não existem restrições no cruzamento de indivíduos em toda a população, ou seja, todo o indivíduo é um possível parceiro de cruzamento (Alba e Tomassini, 2002). Da mesma forma, qualquer indivíduo pode ser retirado e substituído por um novo (operador substituição ou reposição). Podem-se considerar duas classes de **AE** panmícticos, em que a diferença entre elas é a granularidade no passo reprodutivo (Alba e Tomassini, 2002):

- Modelo Geracional (*Generational Model*) – neste modelo, é criada toda uma população que vai substituir a população antiga, todos os indivíduos;

- Modelo Estado-Firme (*Steady-State Model*) – quando aplicado, apenas um ou dois novos indivíduos são criados, a cada iteração, sendo inseridos na população e coexistindo com os indivíduos pais;

Entre estas duas classes existe uma região, um meio-termo, chamado *Generational Gap* onde se encontram modelos em que apenas uma percentagem dos indivíduos são

substituídos. Podemos dizer que o modelo geracional e o modelo estado-firme são subclasses dos modelos da *Generational Gap* (Alba e Tomassini, 2002). A figura 3.2 mostra a posição de cada uma das classes, em que μ é o tamanho total a população antiga e λ o número de indivíduos a serem substituídos. No modelo geracional, nota-se que o número de indivíduos a serem substituídos são todos os da população antiga, ou seja a população será totalmente substituída por uma nova, daí que $\lambda = \mu$.

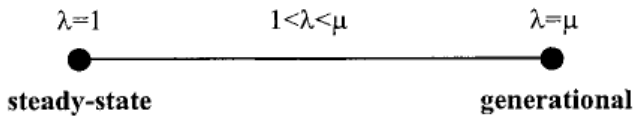

Figura 3. 2 – Algoritmo Panmictico: desde a classe Steady-State até classe generational – retirado de (Alba e Tomassini, 2002).

AE panmícticos são usualmente implementados de forma sequencial, embora seja possível implementá-los de forma a serem executados com paralelização e com ganho no desempenho. O modelo (ou a versão) com paralelismo consiste em executar a função de avaliação sobre cada indivíduo paralelamente. Nos AE, a avaliação do *fitness* de um indivíduo é a operação mais custosa. Isto significa que, se pudermos fazê-lo de forma paralela, isso irá reduzir significativamente o tempo de execução (Alba e Tomassini, 2002).

O resto do algoritmo é executado sequencialmente, uma vez que não compensa paralelizá-lo ou não há ganhos significativos em fazê-lo. Este modelo paralelo de algoritmo panmíctico chama-se paralelismo global (*Global Parallelism*).

3.2 Algoritmos Estruturados

Em **AE** estruturados, ao contrário dos modelos panmícticos, as populações estão divididas espacialmente ou existem restrições no cruzamento entre indivíduos. Apesar de poderem ser executados sequencialmente, são mais adequados para paralelismo. Existem dois modelos principais de **AE** estruturados (Alba e Tomassini, 2002).

- Modelo de ilhas ou **AE D**istribuído (**AED**) – é um modelo de múltiplas populações entre as quais existem esporadicamente alguns cruzamentos entre indivíduos das diferentes populações. Os cruzamentos são mais frequentes entre indivíduos de uma mesma ilha. Este é um modelo mais adequado para ser executado em sistemas que usem memória distribuída, com arquitectura **MIMD** da taxonomia de Flynn (Flynn, 1972) (ver secção 4.2). Normalmente quando este modelo é executado, é indicado que tipo de política de migração se pretende

implementar: topologia da ilhas, quando ocorre migração, que indivíduos são trocados e sincronização entre sub-populações (Belding, 1995), (Tanese, 1989).

- Modelo **AE C**elular (**AEC**) – neste modelo, cada indivíduo tem o seu próprio conjunto de potenciais indivíduos com quem se pode cruzar. Este conjunto é definido pelos indivíduos que se encontram na sua vizinhança. Como tal, cada indivíduo pode pertencer a vários conjuntos de outros indivíduos. Desta forma, o *loop* reprodutivo é realizado em cada um dos numerosos conjuntos existentes, ou seja, um *loop* por cada indivíduo da população. Dado que são executados vários *loops* este modelo torna-se, portanto, mais viável para ser executado de forma paralela. Pode ser executado numa arquitectura **MIMD**, embora seja mais adequado ser executado numa arquitectura **SIMD** (ver secção 4.2).

Existem diferenças entre os modelos acima: o modelo de ilhas possui subpopulações mais numerosas do que no modelo Celular, que por norma apenas possuem um único indivíduo. No entanto, no modelo distribuído, as subpopulações são fracamente acopladas (*loosely coupled*), dado que só esporadicamente existem cruzamentos entre indivíduos de subpopulações diferentes, e estes cruzamentos são aleatórios (embora seja definido na política de migração que tipo de indivíduos se pretende que se cruzem), enquanto que no modelo celular as ligações são fortemente acopladas (*tightly coupled*) dado que cada indivíduo só se cruza com indivíduos que se encontrem na sua vizinhança. Outra diferença é que o número de subpopulações no modelo distribuído é muito menor do que o número de subpopulações no modelo celular. A figura 3.3 mostra a diferença entre os vários modelos da **AE.**

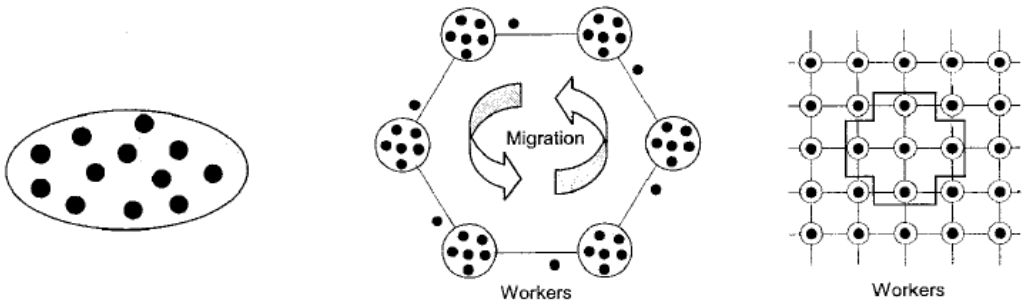

Figura 3. 3 – Vários **AE**: a) **AE** panmíctico, que considera apenas uma única população onde todos os indivíduos se podem cruzar entre si, b) modelos de ilhas, com 6 subpopulações, topologia e política de migração e c) com o modelo celular onde está seleccionado o conjunto de indivíduos com quem a subpopulação (o indivíduo no centro da grelha) se pode cruzar - retirado de (Alba e Tomassini, 2002).

3.3 JECoLi

A **J**ava **E**volucionary **Co**mputation **Li**brary (**JECoLi**) é uma plataforma, implementada na linguagem Java, que implementa alguns dos algoritmos apresentados nas secções anteriores (Evangelista et al, 2009). Existem várias ferramentas que implementam estes algoritmos evolucionários. No entanto, nenhuma delas possui o conjunto de requisitos pretendidos para este trabalho, nomeadamente robustez (*robustness*), modularidade e garantia de qualidade no processo de desenvolvimento de componentes baseados em **CEG,** em aplicações e sistemas onde se pretenda usar estes algoritmos. Estas qualidades são importantes, na medida em que se pretende usar a biblioteca a desenvolver para estender o comportamento actual – permitir a execução paralela dos algoritmos. Uma outra vantagem é que foi desenvolvida com o intuito de permitir uma análise comparativa eficiente entre abordagens distintas em tarefas de optimização específicas (Evangelista et al, 2009).

3.3.1 Arquitectura JECoLi

Na Figura 3.5 podemos ver a arquitectura do **JECoLi**, através do diagrama de classes, e logo a seguir é feita a descrição das entidades mais importantes e qual a sua função dentro da aplicação (Evangelista, Maia e Rocha, 2009):

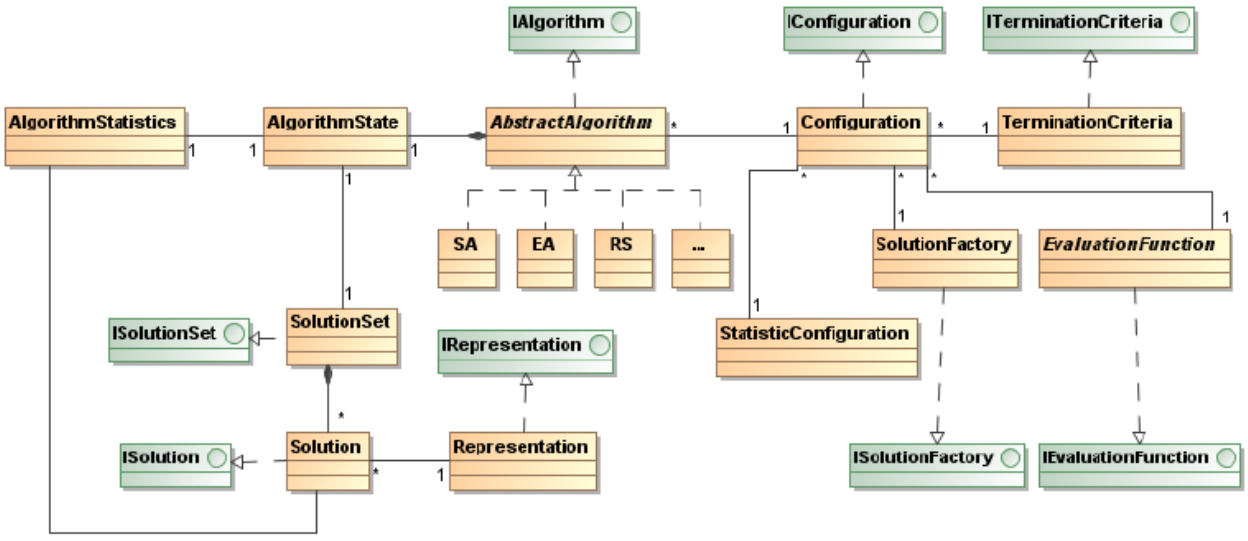

Figura 3. 4 – Arquitectura da plataforma JECoLi (Evangelista, Maia e Rocha, 2009).

- *Algorithm* – representa a abstracção do método de optimização – todos os algoritmos implementados partilham um conjunto de métodos que são definidos na interface *IAlgorithm.*
- *Termination Criteria* – verifica se as condições de terminação são satisfeitas. Está disponível a opção de criar novos critérios de terminação.
- *Evaluation Function* – representa as classes que tratam da descodificação de genomas, transformando em solução para os problemas apresentados. Esta componente está dependente do problema, na medida em que usa os vários algoritmos implementados para resolver o problema. No fundo faz a ligação entre o domínio do problema e o domínio dos algoritmos.
- *Algorithm Configuration* – contêm toda a informação necessária para executar os diferentes algoritmos e pode ser dividida em duas componentes separadas:
    - Informação comum a todos os algoritmos – refere-se a critérios de terminação, função de avaliação e configuração de estatísticas (nomeadamente métricas a serem usadas e calculadas, o que é mostrado ao utilizador e o que é guardado em disco).
    - Informação dependente do algoritmo: refere-se aos diferentes parâmetros dos algoritmos, ou seja, cada algoritmo necessita de um conjunto de parâmetros, dependendo da sua implementação e que não são, claro, comuns.
- *Solutions and solution sets* – cada solução é composta por um genoma, que está codificado numa representação específica, e por um conjunto de valores *fitness* (um por cada objectivo). A parte do conjunto de soluções compreende populações e arquivos.
- *Representations* – classes que implementam uma representação da solução.
- *Solution Factories* – usada para construir e copiar soluções.

Houve igualmente preocupação da parte dos criadores, de desenhar a plataforma de tal forma que futuramente pudesse ser estendida, aproveitando assim as tecnologias de processamento paralelo, permitindo tomar vantagem de ambientes *muticore*, *cluster* e *grid* (Evangelista et al, 2009), algo que se sucedeu mais tarde.

3.4 ParJECoLi

A plataforma **ParJECoLi** (**Par**allelizing **JECoLi**) (Pinho, Sobral e Rocha, 2012) pode ser considerada como a versão do JECoLi que foi adaptada para aproveitar as vantagens que advêm da computação paralela. Esta adaptação foi feita recorrendo ao AspectJ, uma linguagem baseada no modelo POA, devido a natureza dos *concerns* de computação paralela, que são essencialmente transversais. Esta adaptação foi bem-sucedida na medida em que foi

possível executar algoritmos evolucionários retirando vantagens em termos de desempenho e qualidade de soluções nos casos de estudo que foram testados (Pinho, Sobral e Rocha, 2012). Um outro ponto que menciona o sucesso da adaptação, segundo (Pinho, Rocha e Sobral, 2010) e referido de novo em (Pinho, Sobral e Rocha, 2012), tem a ver com a abordagem utilizada, que é referida como não-invasiva, localizada e modular. A utilização desta abordagem permitiu a grande vantagem de a ferramenta JECoLi continuar a ser optimizada – com novos algoritmos, problemas e representações - e, ao mesmo tempo, os módulos de paralelização – aspectos – pudessem ser desenvolvidos, permitindo que a plataforma corra em diferentes cenários. O facto de os *concerns* de computação paralela, com POA, poderem ser representados como aspectos permite que a plataforma se adapte a qualquer ambiente, bastando para isso remover ou adicionar os aspectos que forem necessários consoante o caso[1].

3.4.1 Estratégias de Paralelização

Na secção 3.2 foram apresentados os algoritmos estruturados, cuja principal característica é o facto de serem mais adequados para ser executados em ambientes de computação paralela, embora também possam ser executados sequencialmente. Estes algoritmos estruturados dão-nos uma ideia sobre em que ambientes podem ser executados pela sua configuração e forma de funcionamento. Por exemplo, o modelo de ilhas é claramente mais favorecido se for executado num modelo de memória partilhada ou distribuída, numa arquitectura *multicore* ou *cluster*. O motivo é porque cada ilha tem a sua própria população, o seu próprio algoritmo de evolução e de avaliação de indivíduos, ou seja, teria mais proveito cada um tivesse um processador dedicado a cada ilha – embora na verdade, o que tem é uma *thread* dedicada que será executada pelo processador - e depois, em termos de migração, tanto faria sentido memória partilhada ou memória distribuída, dependendo da arquitectura em que se pretende executar.

Igualmente foi mencionado em (Alba e Tomassini, 2002), que a avaliação do *fitness* de um indivíduo é a operação mais custosa, e que se pudesse ser feita de forma paralela, tal irá reduzir significativamente o tempo de execução. Logo uma forma de paralelizar um modelo que possui apenas uma população, seria dividir a carga da função de avaliação por diferentes *threads* que vão executar em processadores diferentes, reduzindo assim o tempo de execução.

3.4.2 Estratégia de Paralelização no ParJECoLi

[1] o termo que descreve bem esta característica da plataforma é o de ser *pluggable* – propriedade de remover ou adicionar módulos consoante a necessidade, trazendo com isso novas funcionalidades à plataforma e adaptando-a as necessidades

A figura embaixo mostra todas as possíveis configurações que se podem utilizar correndo o ParJECoLi:

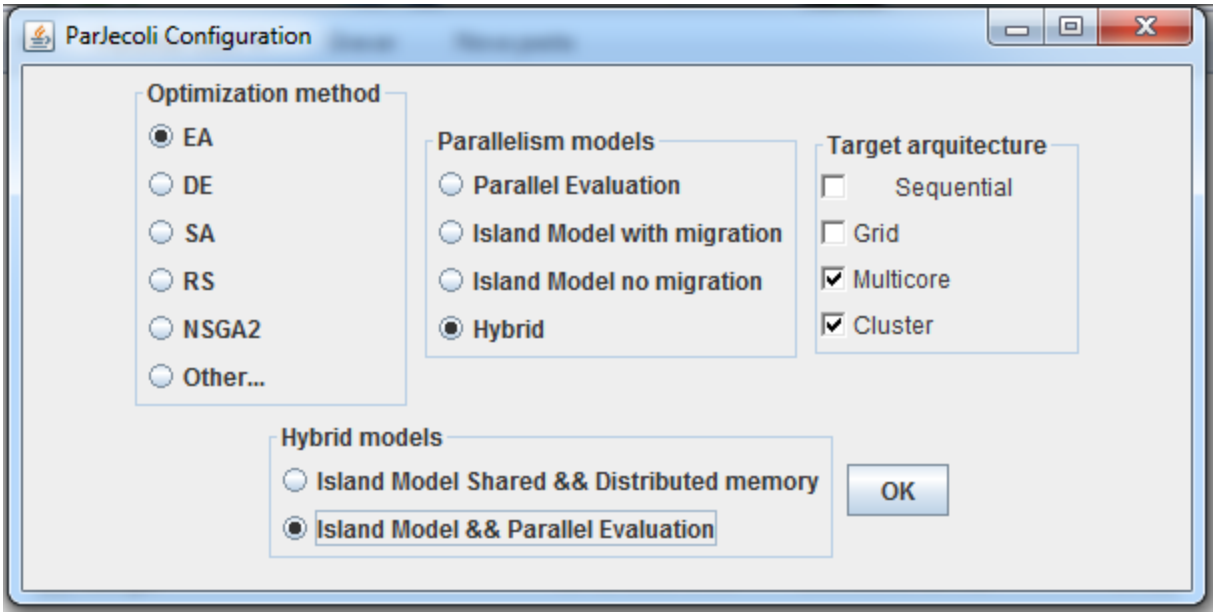

Figura 3. 5 - Janela de configuração do ParJECoLi

Para além de permitir ao utilizador escolher que algoritmo se pretende usar, o utilizador pode escolher qual o modelo de paralelismo, arquitectura alvo em que irá correr, e consoante as escolhas, pode ou não seleccionar modelo híbrido (Modelo de ilhas usando memória distribuída e partilhada e Modelo de ilhas com a função de avaliação a ser executada com paralelismo). Consoante as opções, vão ser seleccionados e carregados os aspectos necessários, que implementam os *concerns* de paralelização para cada caso.

Para este trabalho, foram testadas (e emuladas) apenas as seguintes combinações (por uma questão de simplicidade, iremos considerar que apenas o algoritmo evolucionário é utilizado – EA), sendo que na secção 4.2.1 é descrito com mais pormenor cada arquitectura suportada pelo ParJECoLi:

- O modo sequencial, sem nenhum tipo de paralelismo;
- O modelo *Parallel Evaluation* (em que o trabalho da função de avaliação é dividido por várias *threads*), onde não são permitidos modelos híbridos. É possível seleccionar a opção de executar este modelo numa arquitectura sequencial (apenas um processador) ou *multicore* (vários processadores). Para qualquer dos casos, o funcionamento é igual, ou seja, é gerado um número de *threads*, conforme especificado no ficheiro de configuração;
- O modelo de ilhas com migração, executado numa arquitectura *multicore*;

4. Computação Paralela

Devido ao aparecimento de novos processadores, a computação paralela tornou-se uma das áreas mais conhecidas e investigadas dentro da Engenharia Informática. Não sendo, ainda assim, uma nova área, existe continuamente um grande interesse em obter novas técnicas e métodos que permitam as aplicações obter o melhor desempenho possível, optimizando ao máximo os recursos existentes, obtendo o melhor desempenho possível e aproveitando a grande capacidade de processamento para resolver problemas cada vez mais exigentes.

A Computação Paralela pode ser vista como uma forma de computação em que as instruções de um dado programa, em vez de serem executadas sequencialmente, são executadas paralelamente, ou seja, ao mesmo tempo (Barney, 2014). Parte do princípio que um dado problema pode ser dividido em problemas mais pequenos, que depois vão ser solucionados paralelamente. Podemos considerar diferentes formas de computação paralela: ao nível de bits, ao nível de instruções, dados e tarefas. No caso deste trabalho, tendo em conta o domínio do problema, o interesse em computação paralela pode surgir ao nível de dados ou tarefas (ver secção 4.2).

O termo *thread* é utilizado quando se refere a um fluxo de execução de um dado processo. Usualmente, um processo tem um *main thread*, o fio de execução principal, com um dado número de instruções e um conjunto de valores. Um processo pode conter mais do que uma (são geradas durante a execução), sendo que cada uma tem estado e um conjunto de instruções à sua responsabilidade. Uma importante distinção entre *threads* e processos está no facto das *threads* de um processo partilharem a mesma zona de memória (i.e. as mesmas variáveis) enquanto que os processos requerem mecanismos de comunicação inter-processos – mais pesados – para trocar informação.

Para conseguir a divisão de tarefas, recorre-se ao uso de várias *threads*,dividindo a carga de trabalho por cada uma. A ideia é dividir parte de execução de um processo pelas várias *threads* e, se tivermos uma arquitectura com vários processadores, cada um ocupa-se com uma (o que num sistema ideal, sem outros processos com *threads* a requerer tempo de execução, com o código totalmente paralelizável, o ganho seria proporcional ao número de processadores). O uso de *threads* levanta alguns problemas de desenho, na partilha de recursos, como o acesso à memória partilhada (se estivermos a falar de uma arquitectura de memória partilhada) ou o acesso a disco. Muitas vezes é necessário utilizar mecanismos de

sincronização, que garantem acesso exclusivo e garantem que os dados não são modificados por outra *thread*, levando a resultados de computação incoerentes e falhos.

A execução pode ser optimizada com recurso ao tipo Future, uso de actores (este último mais presente em Scala) e outros mecanismos, dependendo do caso.

4.1 Lei de Amdahl´s

Em teoria, o ganho (referido muitas vezes como o *speedup*) em paralelizarmos uma aplicação sequencial deveria de ser linear ou proporcional, ou seja, consoante o número de processadores que utilizarmos para executar uma aplicação paralela, em proporção deveria de reduzir o tempo de execução. Por exemplo, se tivéssemos dois processadores, o tempo de execução deveria de ser metade do tempo de execução do alcançado por um só processador. No entanto, é algo que muito raramente acontece, pelo que não é uma forma realista de medir o ganho. Ainda que tenhamos apenas um processo com várias *threads*, e cada uma possuísse um processador dedicado à sua execução, existem outros factores a ter em conta. A lei de Amdahl surge como alternativa de medição, e estabelece que o *speedup* máximo que se pode alcançar, sem ter em conta factores adversos, encontra-se limitado pela porção de código que não é paralelizável. Se tivermos uma parte de código que é executado sequencialmente, esta vai sempre limitar o desempenho.

4.2 Taxonomia de Flynn

A arquitectura alvo da execução de um programa paralelo, influencia o modo como o programa vai ser desenhado. Actualmente consideram-se quatro tipos diferentes de arquitecturas de computadores (Flynn, 1972), as quais são classificadas sobre duas dimensões: fluxo de instruções e quantidade de dados (um ou vários conjuntos).

- **SISD** (**S**ingle **I**nstruction, **S**ingle **D**ata) – Existe apenas um fluxo de execução e que processa apenas um conjunto de dados. Este tipo de execução é sequencial, execução feita por apenas um processador, pelo que não é considerado dentro da Computação Paralela.
- **SIMD (S**ingle **I**nstruction, **M**ultiple **D**ata**) –** Uma mesma instrução é aplicada ao mesmo tempo a conjuntos de dados diferentes, existindo paralelismo apenas no conjunto de dados. Exemplos deste tip ode arquitectura são as máquinas vectoriais, GPGPU´s e outros.
- **MISD (M**ultiple **I**nstruction, **S**ingle **D**ata**) –** fluxo de instruções diferentes são aplicados ao mesmo conjunto de dados.

- **MIMD (M**ultiple **I**nstruction, **M**ultiple **D**ata**) –** vários fluxos de execução operam sobre vários conjuntos de dados em simultâneo. Podemos considerar duas subcategorias: arquitectura com memória partilhada e arquitectura com memória distribuída.

Nalguns casos, em vez de se chamar instruction (instrução) o termo é substituído por program (programa) (Barney, 2014). Na prática, representa o mesmo: um programa é um conjunto de instruções, se tivermos vários conjuntos, temos vários programas.

Considera-se arquitectura de memória partilhada uma máquina com vários processadores que partilham a mesma memória física (Barney, 2014). Consideram-se duas categorias ou dois modelos dentro desta arquitectura: SMP (Symmectric MultiProcessors, noutros tempos era referido como UMA – Uniform Memory Acess), na qual os vários processadores ou cores estão ligados a um bus partilhado e acedem e usam uma só memória, com um tempo de acesso uniforme, e as NUMA (Non-Uniform Memory Acess), em que cada processador tem a sua memória privada, embora possa aceder à memória dos outros (Barney, 2014). O ambiente multicore, presente em grande maioria dos computadores, é baseada nesta arquitectura.

As vantagens desta arquitectura é que é a partilha de dados entre tarefas (entre threads diferentes) é mais fácil e uniforme devido à proximidade da memória com os processadores. Uma das principais desvantagens é exactamente a necessidade de o programador ter de desenhar bem a sincronização e partilha à memória. Geralmente uma concepção inapropriada leva a problemas na execução.

As arquitecturas de memória distribuída correspondem a arquitecturas em que vários nós estão ligados entre si, sendo que cada nó pode conter vários processadores e a sua própria memória. Um exemplo desta arquitectura são os clusters, (Barney, 2014), os quais necessitam de um mecanismo alternativo para comunicar entre si, dado que não partilham de uma zona de memória comum. As abordagens usualmente utilizadas são **PVM** (**P**arallel **V**irtual **M**achine) (PVM, 2014) e o **MPI** (**M**essage **P**assage **I**nterface) (Groop et al, 1996). Tanto uma como a outra disponibilizam mecanismos para a troca de mensagens entre processos, embora de formas diferentes.

4.2.1 Arquitecturas suportadas pelo ParJECoLi

No PArJECoLi, está previsto o uso de arquitectura sequencial, *cluster*, *multicore* e *grid* (Barney, 2014). Embaixo é descrita cada uma das arquitecturas e a diferença entre si:

- Sequencial – arquitectura onde está presente apenas um processador. É possível dividir o trabalho por várias *threads*, algo que o ParJECoLi faz quando divide o trabalho da função de avaliação. No entanto, não existem ganhos significativos, na medida em que as *threads* têm de ser executadas pelas mesmo processador;

- Multicore (múltiplos núcleos) – é uma arquitectura onde um processador pode ter vários núcleos (actualmente, 2 ou mais), que na prática funcionam como um processador independente e *cache* própria, embora os núcleos, em algumas arquitecturas, partilhem a mesma *cache*. É interessante considerar que podemos estar presente de uma arquitectura com memória partilhada (se partilharem a mesma *cache* e/ou usarem uma comunicarem, ou arquitectura de memória distribuída, na qual comunicam entre si através da troca de mensagens;
- Cluster (agregado de computadores) - como o nome indica, é uma arquitectura em que vários computadores estão ligados entre si, podendo essa ligação ser forte ou fraca. Nesta arquitectura, cada nó é um computador, na maioria dos casos com o mesmo *hardware* e sistema operativo. A ideia desta arquitectura é ligar vários nós para funcionarem como um só. Ou seja, de um ponto de vista exterior, considerando apenas o seu funcionamento, seria semelhante a um computador com vários processadores, cada um com a sua própria *cache*, que executam a mesma tarefa, normalmente controlada e gerida por um dos nós. Num *cluster*, nem sempre os nós são homogéneos, ou seja, é possível ter nós com características diferentes, embora se torne mais difícil analisar e medir o desempenho.
- Grid – é uma arquitectura cujo princípio é bastante semelhante ao *cluster*, mas com grandes diferenças. Semelhante no sentido em que se pretende ligar várias máquinas para realizarem tarefas, obtendo assim grande capacidade de processamento. Diferente, no entanto, no funcionamento e configuração. Ao contrário do *cluster*, os computadores não precisam estar no mesmo local físico, normalmente os nós realizam tarefas diferentes, são heterogéneos e estão fracamente acoplados entre si. Podemos ver a *grid* como uma estrutura onde vários recursos se podem conectar para realizar uma dada tarefa - um exemplo, SETI Project (SETI, 2014);

4.2.2 Futures

É um mecanismo de concorrência (Lea, 1999), criado com o intuito de permitir que uma dada *thread* continue a sua execução, sem ficar à espera do resultado da computação de um método numa outra *thread*. Esta apenas é bloqueada quando tenta usar o valor antes da computação ser terminada. É um mecanismo que permite invocações assíncronas em dois sentidos. Quando a computação é terminada, a variável do tipo Future, vai ser modificada e passa a conter o resultado daquela. A ideia deste mecanismo é permitir que a *thread* continue a sua execução, até necessitar de utilizar o valor, servindo o Future como "garantia" que irá receber o resultado da computação, permitindo assim um desempenho mais optimizado.

5. Scala

O Scala é uma linguagem de programação multiparadigma, desenvolvida em 2001 (primeira versão) pelo Professor Martin Odersky, da Universidade de Lausanne, Suiça (Scala, 2013). O seu nome, Scala, deriva do facto de ser considerada uma linguagem *escalável*, ou seja, é uma linguagem que foi construída para satisfazer as exigências dos utilizadores, seja o seu intuito criar pequenos *scripts* ou desenvolver grandes aplicações (e possivelmente estendê-las). Pode-se dizer que cobre uma grande variedade de tarefas de programação, desde as mais simples até as mais complexas (Odersky et al, 2008). É uma linguagem que assenta em dois paradigmas de programação:

- **P**rogramação **O**rientada a **O**bjectos (**POO**) (ver início do Capítulo 2)

- **P**rogramação **F**uncional (**PF**) – é um modelo de programação que é orientado por funções. No fundo, o modo de programar é usando funções, não guardado estado dos dados.

A fusão entre estes dois paradigmas permite usar os pontos fortes de ambos de forma a complementarem-se, permitindo o ponto forte desta linguagem que é a escalabilidade. Os mecanismos de programação funcional permitem construir pequenos programas, de forma simples e eficiente, enquanto que os mecanismos de programação orientada a objectos permitem a construção de grandes sistemas, de forma estruturada, e a fácil extensão dos mesmos se pretendido (Odersky et al, 2008).

Uma das vantagens desta aproximação é que quando se desenvolve uma aplicação grande, é possível integrar estes dois paradigmas: utilizando os mecanismos do POO para estruturar a aplicação e torna-la robusta e extensível e, usar os mecanismos de PF para resolver os problemas mais pequenos de forma simples e eficaz (Odersky et al, 2008). A fusão destes dois paradigmas abre portas para novos estilos de programação.

Uma característica desta linguagem é que igualmente foi desenhada para simplificar, ou seja, foi desenhada de tal modo que é possível desenhar *scripts*, que noutras linguagens ocupariam bastantes linhas, por vezes de forma pouco ortodoxa e o Scala apresenta mecanismos e técnicas que promovem a eficiência e resolvem algumas limitações detectadas noutras linguagens (Odersky et al, 2008).

5.1 Vantagens na utilização do Scala

Esta linguagem apresenta muitas vantagens para o trabalho que se pretende realizar. Uma delas é claro a escalabilidade: ela foi desenhada para dar suporte a grandes sistemas.

Uma outra é que permite interoperabilidade com a plataforma Java. Ou seja, o compilador do Scala reconhece e compila código fonte Java (Scala, 2013), o que permite, por exemplo, usar a aplicação JEcoLi, totalmente programada em Java, sem ser necessário fazer qualquer alteração para ser reconhecida, se necessário. A linguagem Scala reconhece métodos Java, acede a campos Java, implementa interfaces em Java e pode herdar classes em Java, sem qualquer problema. De facto, muitas das suas bibliotecas são do Java. Aliás, segundo (Odersky et Al, 2008), o Scala reutiliza vários tipos do Java – os *Ints* do Scala são representados usando o tipo primitivo *int*, a mesma coisa para *floats*, *booleans* e muitos outros. Não só os reutiliza como ainda os melhora. Apesar de usar muitas das bibliotecas do Java, apresenta os seus próprios mecanismos (Odersky et Al, 2008) - alguns deles são melhorias dos mecanismos já existentes em Java. Uma das melhorias é aquilo a que o autor se refere como *syntatic sugar*, ou seja, a forma como foi desenhada permite ao programador utilizar uma sintaxe muito simples e conveniente, levando a uma redução de linhas de código (Odersky et Al, 2008). No entanto, a utilização do Scala em Java por vezes causa limitações nos mecanismos do primeiro, dado que a forma como o último está implementado não permite explorar o potencial dos mecanismos. Mais abaixo serão explicadas e demonstradas essas limitações, assim como no capítulo 7.

Dado que o objectivo do trabalho é estudar a capacidade da linguagem no suporte à computação paralela, baseando-nos nos princípios da POA – especificamente a sua capacidade de suporte e composição de módulos - uma das coisas que foi alvo de pesquisa é se realmente esta linguagem possui mecanismos para os implementar. A secção a seguir apresenta alguns das técnicas que são usadas nas soluções apresentadas, para melhor compreensão das mesmas.

5.2 Mecanismos do Scala

Esta secção serve para apresentar alguns dos mecanismos do Scala. Dado que existem muitos aspectos do Scala, seria tarefa pouco viável apresenta-los a todos. Como tal, será feita a apresentação dos mecanismos que são usados nalgumas das soluções apresentadas no capítulo 7.

5.2.1 Traits

> "*Traits são uma unidade fundamental na reutilização de código em Scala*", retirado de (Odersky et al, 2008).

Um *Trait* é um módulo como uma interface, uma classe, e que, tal com estes, permite composição. Possui uma sintaxe em tudo parecida ao da classe (ver Figura 5. 1), no entanto apresenta diferenças significativas que torna este mecanismo uma poderosa ferramenta, no que toca a reutilização de código em Scala (Odersky et al, 2008). Encapsula métodos e membros privados, tal e qual uma classe, podendo ser depois reutilizada por uma classe, através de uma técnica chamada *mixin* (Odersky et al, 2008). Comparando com o Java é equivalente a uma interface mas muito mais poderoso, uma vez que pode conter estado e comportamento (para modificar o estado e/ou para dar a conhecê-lo).

```
trait Philosophical {
  def philosophize() {
    println("I consume memory, therefore I am!")
  }
}
```

Figura 5. 1 – Exemplo de um *Trait* (Odersky et al, 2008).

Uma vez definido, pode ser misturado (*mixed in*) com uma classe usando a palavra *extends* (semelhante ao *extends* do Java) ou *with* (ver Figura 5. 2). Uma classe pode ser misturada com mais do que um *Trait* (tal e qual como em Java se pode implementar mais do que uma interface). Tanto a classe com o *Trait* pode ser misturado com mais do que um *trait*, mas não mais do que uma classe.

```
class Frog extends Philosophical {
  override def toString = "green"
}
```

Figura 5. 2 – Classe *Frog* mistura com o *Trait Philosophical*

Tal como em múltipla herança, a classe *Frog* também herda os métodos de *Philosophical* e, da mesma forma, pode fazer *override* dos mesmos. Um *Trait* também pode ser usado como um tipo. Neste caso podíamos definir uma variável do tipo *Philosophical*. Ao contrário de uma interface do Java, possui membros privados e é possível manter o estado e, apesar de ter

sintaxe quase igual ao de uma classe, não possui construtor e como tal, não é possível fazer *new* (ver Figura 5. 3).

```
val phil : Philosophical = new Philosophical
val phil1 : Philosophical = new Frog
```

Figura 5. 3 – Compilador não reconhece construtor para *Philosophical*. Uma variável que seja do tipo de um *Trait* pode ser inicializada por uma classe que esteja misturada com aquele (neste caso, a classe *Frog*).

Existe uma diferença em relação à técnica de múltipla herança, utilizada noutras linguagens **POO.** A diferença é a linearização. Segundo (Odersky et al, 2008), quando uma classe é instanciada, o compilador do Scala pega na classe e em todos as classes e *traits* herdados e ordena-os de uma forma simples e linear. Quando é chamado o operador *super* dentro dessas classes, o método invocado é da classe (ou *trait*) a seguir na ordem. Se todos os que estão na ordem, excepto o último, chamarem *super*, então vamos ter um comportamento em pilha. Na linearização, a classe é sempre linearizada primeiro antes das superclasses e *traits* misturadas. Como tal, sempre que uma dada função usa *super*, esta vai modificar as funções que estão acima dela e não o contrário. A seguir temos um exemplo que mostra as propriedades de linearização em Scala.

```
class Animal
trait Furry extends Animal
trait HasLegs extends Animal
trait FourLegged extends HasLegs
class Cat extends Animal with Furry with FourLegged
```

Figura 5. 4 – Classes e *traits* em Scala (Odersky et al, 2008)

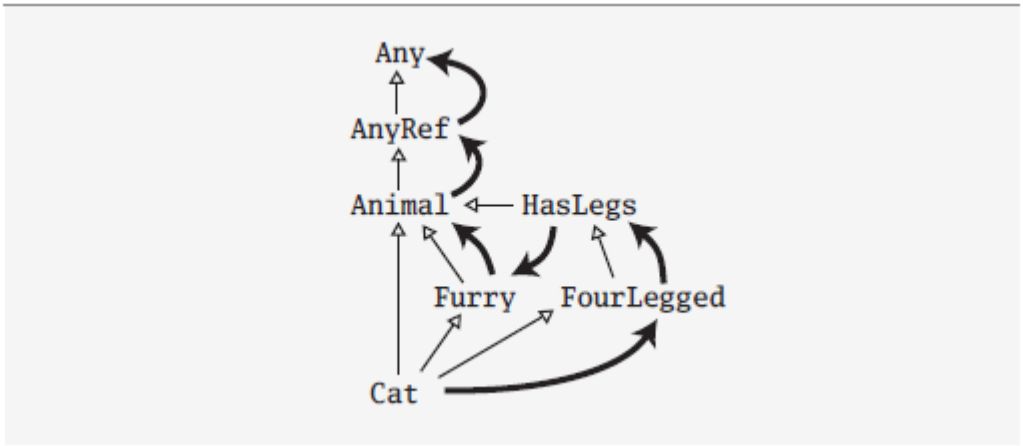

Figura 5. 5 – Hierarquia de herança e linearização para a classe *Cat* (Odersky et al, 2008).

Na Figura 5. 4 – Classes e *traits* em Scala (Odersky et al, 2008), podemos ver que a classe *Cat* herda a classe *Animal* e os *traits Furry* e *FourLegged* (*Cat extends Animal with Furry with Fourlegged*). A Figura 5. 5 mostra a hierarquia entre as várias classes e *traits* do exemplo, para melhor compreensão das ligações. Note-se que as setas brancas indicam a ordem de herança, enquanto que as setas a preto indicam a ordem de linearização. Podemos ver que quando *HasLegs* chama o operador *super* numa função, a chamada vai ser feita no *Trait Furry*. Por herança, no entanto, *HasLegs* herda os métodos de *Animal*. A chamada de *super*, neste caso, só funciona porque *Furry* também herda os métodos de *Animal*. Um pormenor apenas na Figura 5. 5: a classe root é *Any*, que tem por subclasse *AnyRef*. É a classe base de todas a classes de Scala (incluindo *Traits*). A classe *AnyRef* é apenas um outro nome para *java.lang.Object* (Odersky et al, 2008). Ver (Macbeath, 2014) para mais exemplos.

As classes em Scala podem implementar interfaces Java e métodos em classes Java podem receber e utilizar instâncias de classes Scala que implementem essa interface.

5.2.2 Traits em Java

Na linguagem de programação Java, nas versões estáveis actualmente existentes que são disponibilizadas, não existe mecanismo *Trait*. Isto faz com que não seja possível utilizar todas as capacidades que este apresenta, numa classe Java, da forma natural que acontece quando usamos uma classe Scala. Ainda assim é possível obter melhorias ou vantagens oferecidas pelos *traits* que não são possíveis de obter com interfaces (Spiewak, 2013). Para uma classe em Java poder implementá-lo, o compilador do Scala transforma-o numa interface, ao nível do *bytecode* (Spiewak, 2013). Os exemplos (Figura 5. 6, Figura 5. 7, Figura 5. 8, Figura 5. 9, Figura 5. 10, Figura 5. 12) mostram a implementação de um *trait* numa classe Java, qual a semelhança com interfaces e quais são as vantagens que aqueles oferecem às classes Java:

```scala
trait Model {
  def value: Any
}
```

Figura 5. 6 - Exemplo de um *Trait* em Scala com um método abstracto (não definido) (Spiewak, 2013).

```java
public interface Model {
    public Object value();
}
```

Figura 5. 7 – Interface equivalente ao *trait* (Spiewak, 2013).

Para uma classe Java, que implementa o *Trait* na Figura 5. 6, não existe diferença para uma interface. Ao nível de *bytecode*, um *Trait* é semelhante a uma interface, o que significa que podemos dizer que o *Trait* na Figura 5. 6 é equivalente à interface em Java na Figura 5. 7. Embaixo um exemplo de uma classe que implementa um *Trait*:

```java
public class StringModel implements Model {
    public Object value() {
        return "Hello, World!";
    }
}
```

Figura 5. 8 - Implementação do *Trait* Model numa classe Java (Spiewak, 2013).

De notar que o *Trait*, na Figura 5. 6, não possui membros privados e método não está definido, apenas está declarado (como numa interface). Na Figura 5. 8, é implementado o *Trait* e o método declarado é definido, tal e qual como se implementasse uma interface. Podemos ver que o resultado é o mesmo se misturarmos uma classe Scala com este mesmo *Trait*. Quando se pretende usar todas as suas capacidades, no entanto, a situação complica-se um pouco mas também mostra a diferença de capacidade entre um e outro mecanismo (Spiewak, 2013). Para mostrar a forma como o Scala "resolve" este problema, acrescentamos um novo método chamado `printValue()` ao *Trait* Model, conforme está na Figura 5. 9:

```scala
trait Model {
  def value: Any

  def printValue() {
    println(value)
  }
}
```

Figura 5. 9 - Acrescentado o método `printValue()` (Spiewak, 2013).

```java
public interface Model extends ScalaObject {
    public Object value();

    public void printValue();
}

public class Model$class {
    public static void printValue(Model self) {
        System.out.println(self.value());
    }
}
```

Figura 5. 10 – Interface Model e classe auxiliar Model$class, ao nível do bytecode (Spiewak, 2013).

Na Figura 5. 10, podemos ver que o que o compilador do Scala faz é criar uma classe auxiliar que contém todos os métodos definidos no *Trait*. Note-se que o método

`printValue()` está presente nesta classe mas o método `value()` não. Tal acontece porque este último está declarado e não definido e, como tal, pode ser apresentado apenas na interface. O método `printValue()` está declarado na interface e definido na classe auxiliar como um método estático. Vamos ver agora dois casos: uma classe Scala a **estender** aquele *trait* - Figura 5. 11 - e qual o resultado ao nível de *bytecode* - ver Figura 5. 12 - e outro caso em que uma classe Java **implementa** o mesmo *trait*. Para Java, como já foi visto, um *trait* é uma interface e para Scala, um *trait* é mais do que uma interface, daí a diferença.

```
class StringModel      extends Model{
    def value:Any = "Hello, World!"
}
```

Figura 5. 11 - Classe Scala estende Model.

```java
public class StringModel implements Model {
    public Object value() {
        return "Hello, World!";
    }

    public void printValue() {
        Model$class.printValue(this);
    }

    // method missing here (see below)
}
```

Figura 5. 12 - Classe Scala depois de compilada (Spiewak, 2013).

Podemos ver na Figura 5. 11 que o método printValue() não está presente na classe. Olhando para os *bytecodes* da classe na Figura 5. 12, o método `printValue()` está presente e é chamado directamente da classe auxiliar criada, com os métodos definidos. O compilador do Scala automaticamente faz a ligação entre o método definido na classe auxiliar e o método na classe estendida. Quando se procura o mesmo efeito numa classe Java, não é possível, uma vez que as classes Java só reconhecem um *trait* como uma interface, mesmo que tenha métodos definidos. Se tentarmos estender uma classe com aquele, ele automaticamente dá erro, avisando que não é uma classe abstracta para que possa estender. Como tal, leva a que seja implementada como uma interface. O problema é que assim ele não detecta o método `printValue()` como definido mas sim como um método declarado e que precisa de implementar. Uma forma de dar a volta a esta situação é usar a classe auxiliar criada pelo compilador do Scala, carregando-a dinamicamente e chamando o método estático (ver Figura 5. 13). O problema é que esta solução não é uma solução natural, que pode levar a problemas caso existam alterações no *trait*.

A empresa detentora dos direitos do Java, a Oracle, disponibilizou uma versão – a versão 8 (ou release 8), o jdk8 – que implementa e disponibiliza mecanismos chamadas interfaces funcionais, cuja ideia é oferecer alguma das funcionalidades dos *traits*, como por exemplo, permitir a definição de uma função logo na interface (JDK8, 2014). Ainda assim, depois de testar e estudar o seu funcionamento e interoperabilidade com Scala, verificou-se que não

trouxe melhorias ou vantagens nessa interacção, continuando-se a verificar os mesmos problemas.

```
public class StringModelJava implements Model{

    @Override
    public Object value() {
        printValue();
        return null;
    }

    @Override
    public void printValue() {
        Class<?> cls;
        try {
            cls = Class.forName("Model$class");
            Method methos[] = cls.getDeclaredMethods();
            System.out.println(methos[0].getParameterTypes().length);
            methos[0].setAccessible(true);
            methos[0].invoke(null ,this);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Figura 5. 13 - Classe Java implementa Model

5.2.3 Funções em Scala

Em Scala existe distinção entre métodos e funções (Macbeath, 2014). Um método, tal como em Java, é parte de uma classe, tem nome, assinatura, possivelmente anotações. Uma função, em Scala, é um objecto completo. Em (Odersky, 2008) podemos ver que uma função é um objecto e, como tal, pode ser atribuída a uma variável, pode ser guardada como um valor. A linguagem apresenta vários *traits* que permitem implementar estas – *Function0*, *Function1*,…, *Function22* – sendo que a diferença entre elas é o número de argumentos que permitem a uma função receber (Macbeath, 2014) (Odersky, 2008). Sendo uma função uma instância de uma classe, naturalmente vai ter métodos. Um desses métodos é o método *apply()*, sendo este o que contém o código que implementa o corpo de uma função (Odersky, 2008) (Macbeath, 2014).

```
object teste {
  def method(x: Int) = x + 1                    > method: (x: Int)Int

  val function = (x: Int) => x +1               > function  : Int => Int = <function1>

  val function2 = (x:Int) =>  method(x)         > function2  : Int => Int = <function1>
}
```

Figura 5. 14 - Método e funções

Na Figura 5. 14 podemos ver que um método é declarado usando a palavra `def` e a função, sendo um objecto, é guardada num `val` (é um valor fixo, significa que uma vez atribuído um objecto aquele valor, não pode ser modificado). Podemos ver que cada uma das expressões foi avaliada pelo interpretador do Scala e que deu resultados diferentes (do lado direito de cada expressão está o resultado devolvido pelo interpretador). No primeiro caso, do método, podemos ver o nome do método (*method*), de seguida, dentro de parêntesis, o nome e tipo de cada argumento (neste caso só temos um) e por fim, o tipo que é retornado pelo método, que é *Int* dado que o que ele faz é somar um valor ao argumento que é passado e o resultado dessa soma é um número inteiro. No segundo caso, temos a declaração de uma função e, olhando primeiro para a expressão que foi guardada no valor fixo, vemos que foi declarada usando uma técnica chamada *function literal* (Odersky, 2008). Esta técnica é uma sintaxe alternativa que permite criar funções de forma simples. Do lado esquerdo do operador `=>` estão os argumentos da função, do lado direito o corpo da função. O interpretador do Scala associa a expressão ao nome do valor fixo e, quando queremos chamar a função, basta escrever (baseando-nos no exemplo da Figura 5. 14) `function(1)`, por exemplo. Só uma pequena chamada de atenção para o resultado devolvido pelo interpretador: function é do tipo `Int` (argumento) `=>` `Int` (tipo retornado) que é o mesmo que dizer que `function` é do tipo *Function1*, *trait* que representa funções que recebem um argumento.

Uma característica do Scala é a de permitir ao utilizador escrever código mais simplificado comparado com outras linguagens, pelo uso de uma sintaxe - (Odersky, 2008) considera que possui os atributos que a permitem considerar como *syntatic sugar* - que permite tornar as construções ou definições de código de forma mais natural e simples. Esta característica pode ser vista com este pequeno exemplo: se escrevermos o nome da função seguido de parêntesis (com ou sem argumentos), o interpretador do Scala automaticamente chama o método `apply()` dentro do objecto, com os argumentos passados. Ou seja, pegando no exemplo da Figura 5. 14, se escrevermos `function(1)`, o interpretador automaticamente traduz para `function.apply(1)` (Odersky, 2008)(Macbeath, 2014). Também podemos ver um outro pormenor, deixado de parte anteriormente: na definição do método, ao contrário do que acontece em Java, não foi indicado qual o tipo que retorna. No entanto, podemos ver pela expressão do lado direito, que o tipo retornado é *Int*, conclusão obtida automaticamente pela avaliação do interpretador. Outro, já mencionado, é o uso do *function literal*, uma sintaxe alternativa que permite ao programador criar uma função de forma simples.

5.2.4 Funções Parciais

São dois mecanismos com nomes muito parecidos mas que na prática são diferentes (Andrási, 2013). Em termos matemáticos, dada uma função f:X'→Y, diz-se que uma função é parcial quando está definida para um subconjunto de X'. Um exemplo concreto de uma função parcial é a função de divisão: está definida para todos os números reais excepto o zero, ou seja, para um subconjunto dos números reais (Andrási, 2013). As funções parciais em Scala são implementadas exactamente da mesma forma: apenas lidam com um subconjunto dos argumentos que pode receber (por exemplo, recebe argumentos do tipo int mas só resolve casos de um a dez). Uma função parcial é implementada usando o mecanismo de *pattern matching* do Scala.

5.2.5 Funções de Ordem Elevada (High Order Functions)

High order functions são funções que aceitam como argumento outras funções (mas não métodos). É um mecanismo do Scala que é utilizado numa das soluções em Scala no Capítulo 7 (Odersky et al, 2008). Na Figura 5. 15 temos um exemplo:

```
def filesMatching(query: String,
    matcher: (String, String) => Boolean) = {

  for (file <- filesHere; if matcher(file.getName, query))
    yield file
}
```

Figura 5. 15- Exemplo de uma *High-Order* (Odersky et al, 2008).

Neste exemplo, temos um método – *filesMatching* – que recebe como argumentos *query*, que é uma *string* e *matcher*, um *function literal* e que é equivalente a dizer que é um objecto do tipo *Function2*, que recebe dois argumentos do tipo *string* e devolve um *boolean* (ver Figura 5. 16).

```
val matcher = (a:String, b:String) => true:Boolean   > matcher  : (String, String) => Boolean = <function2>
```

Figura 5. 16 – Avaliação do argumento `matcher` pelo interpretador do Scala

Dentro da função *filesMatching*, é executada então a função dentro de um *if*.

5.2.6 Objectos e membros estáticos

Ao contrário da linguagem Java, que permite criar membros estáticos numa classe – métodos ou variáveis globais à classe – em Scala só existem membros e métodos estáticos dentro de um módulo especialmente criado para isso chamado *object*. Este módulo é a implementação do conceito de *singleton object*, conforme descreve (Odersky et al, 2008):

- É um objecto definido com a palavra `object`;
- Cada objecto tem uma instância apenas e pode ser acedida globalmente (geralmente pelo nome do objecto);
- Um objecto que partilhe o seu nome com uma classe e está definido no mesmo ficheiro que ela, é considerado *companion object* (objecto associado) da mesma. A classe é considerada *companion class* (classe associada);
- Um objecto que não tenha uma classe é considerado um objecto único;

Um objecto pode herdar métodos de classes, *traits* ou até de outro objecto, pode ser passado como um parâmetro e implementar interfaces. Tanto o objecto como a classe podem aceder aos membros privados um do outro.

5.2.7 Actores

Um Actor é um mecanismo, que está presente no Scala e que se apresenta como principal construtor para implementar modelos de paralelismo em programas que se pretende desenvolver. O Scala utiliza o sistema de *threads* do Java, mas introduz o uso de actores, que segundo (Odersky et al, 2008) providenciam modelos de concorrência que facilitam ou resolvem boa parte dos problemas com que os utilizadores se deparam quando usam *threads*. De certa forma, usar actores é como escrever um programa que usa múltiplas *threads* mas sem recorrer ao uso de *locks* ou de variáveis. (Odersky et al, 2008) define um actor como sendo uma entidade parecida ou facilmente identificável como uma *thread*, com *buffer* para receber mensagens. Podemos considerar dois tipos de actor que diferem na forma como esperam por mensagens:

- Temos o actor que actua no modo receptor, o qual gere uma *thread* java.
- Temos o actor que actua no modo reactivo, o qual é na sua essência uma *thread* por CPU, a qual gere vários actores.

```
val echoActor = actor {
	while (true) {
		receive {
		case msg =>
		println("received message: "+ msg)
		}
	}
}
```

Em cima temos o exemplo de um actor que actua como receptor. Possui um método chamado *receive*, que processa mensagem que recebe. A mensagem é processada recorrendo ao mecanismo de *pattern matching* do Scala. Cada caso pode retornar uma mensagem se necessário. Como possui estado próprio, torna-se necessário mudar de *thread* quando um outro actor acorda, sendo por isso necessário uma por cada, o que pode se tornar dispendioso se estivermos a usar bastantes actores. Existe um custo elevado em ter muitas *threads* e mudar entre elas.

No outro caso, onde o actor actua de modo reactivo, permite a reutilização de *threads*, diminuindo assim o custo de mantê-las e de mudar entre elas. Isto é conseguido fazendo com que o actor não retorne nenhum tipo. Desta forma anula-se a necessidade de utilizar a *call stack* da *thread* e a necessidade de mudar, podendo ser reutilizada no próximo actor que acordar (Odersky et al, 2008). Neste caso, é utilizado um método diferente, chamado *react* o qual não retorna nenhum tipo, daí que a *thread* possa ser reutilizada. Em teoria pode-se utilizar somente uma *thread* para servir todos os actores, se todos actuarem de modo reactivo. Embaixo podemos ver um exemplo de um actor que actua neste modo:

```
object NameResolver extends Actor{
	import java.net.{InetAddress, UnknownHostException}
	def act() {
		react {
		case (name: String, actor: Actor) =>
		actor ! getIp(name)
		act()
		case "EXIT" =>
		println("Name resolver exiting.")
		// quit
		case msg =>
		println("Unhandled message: "+ msg)
		act()
		}
	}

	def getIp(name: String): Option[InetAddress] = {
			try {
				Some(InetAddress.getByName(name))
			} catch {
			case _:UnknownHostException => None
			}
	}
}
```

6. Trabalho Relacionado

Nesta secção, são apresentas algumas das alternativas que foram pesquisadas para realizar o trabalho.

6.1 Akka

Akka (Akka, 2014) é um *toolkit* de código aberto, criado por Jonas Bonér e distribuído pela empresa *Typesafe*, com o intuito de ajudar programadores a criar aplicações paralelas e concorrentes de uma forma mais fácil, sem ter de se deparar com os problemas que usualmente se encontra na fase de desenvolvimento – os problemas que advêm de criar *threads* que partilham memória e necessitam sincronização. É um toolkit que suporta o modelo de concorrência baseado em actores.

Segundo (Wyatt, 2012), em aplicações que usam *threads* e partilham um grande número de recursos – levando à necessidade de criar barreiras para partilhar recursos e de haver sincronização entre eles – aparecem muitas vezes problemas e inconsistências que, por mais que o programador tente, nem sempre consegue prever e, muitas vezes, o excessivo uso destes mecanismos para controlar o acesso a recursos causa mais problemas do que os que resolve. Devido a estes motivos, surgiu a necessidade de criar uma *toolkit* que permitisse um uso mais simples e organizada de *threads* – surgindo daí o conceito de actores e, posteriormente, o Akka.

Segundo (Wyatt, 2012), Akka foi desenhado paralelamente ao Scala, utilizando código Scala, o que levou a que existisse alguma cumplicidade entre eles – ambos utilizam os tipos Actor (ver secção 5.2.7) e Future. No entanto, a dado ponto essas estruturas sofreram evoluções diferente e só a partir do Scala 2.10 é que ambos passaram a estar, até certo ponto, alinhados – Scala passou a usar o mesmo tipo de Actor implementado pelo Akka, dado que a estrutura que este implementou provou ser mais eficiente e mais conseguido. Também foi desenvolvido para ser utilizado em Java, ou seja, um programador que queira utilizar este toolkit com código Java tem acesso a interfaces que o permitem utilizar.

6.2 Padrão Visitor

O padrão Visitor é apresentado em (Gamma et al, 1994), como um padrão pertencente à categoria de padrões comportamentais. A intenção do padrão Visitor é o de separar operações da estrutura. Basicamente, quando queremos adicionar uma nova operação, podemos faze-lo sem alterar a estrutura, ou seja, de forma independente.

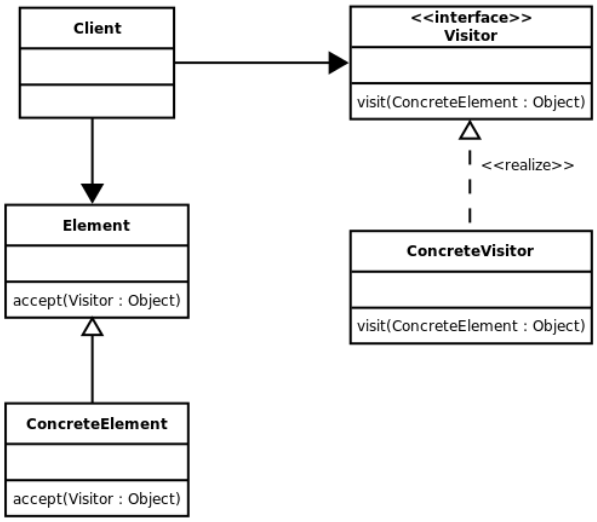

Figura 6. 1 - Diagrama de classes do padrão Visitor

Quando temos uma hierarquia de classes que é grande, pode acontecer ser complicado adicionar uma nova operação ou até uma alteração de código, uma vez que exige que a mesma seja definida ou alterada em todas as classes. Por exemplo, uma alteração numa função implicaria ir a cada uma das classes e alterar a respectiva operação, enquanto que se todas as operações estivessem no mesmo sítio seria mais fácil alterar.

6.2.1 Modo de funcionamento

A estrutura e as operações têm interfaces próprias (Element e Visitor). A interface Element tem um método chamado *accept*, que tem como argumento um elemento do tipo Visitor. Basicamente, este método permite à estrutura receber um visitante que vai realizar uma dada operação sobre o elemento. O visitor chama um método definido na sua classe que recebe como argumento um elemento do tipo Element. Ou seja, um elemento recebe um visitante, através do método accept, que por sua vez invoca um método que recebe como argumento o elemento que aceitou o visitor. Deste modo, o visitor tem acesso às variáveis internas do elemento e pode alterar o seu estado. A ideia seria usar o modo de funcionamento do padrão para emular um funcionamento muito parecido ao do POA. O problema é que exige alterações estruturais, o que faz com que seja muito invasivo e, por isso, faz com que

seja muito difícil implementá-lo, não permitindo, igualmente, depois de realizadas as alterações, reverter a aplicação paralelizada ao estado original. Como tal, apesar da sua simplicidade, não se apresenta como uma solução para implementar os *concerns* de computação paralela, para além de que não é solução para maioria dos casos. A sua introdução neste capítulo serve como referência de que foi explorada esta hipótese.

7. Implementação

Neste capítulo é feita a apresentação e descrita a implementação de duas soluções que foram pesquisadas. Estas duas soluções são diferentes abordagens, em Scala, para conseguir emular o funcionamento da POA. Para conseguirmos testar se estas abordagens são bem-sucedidas, utilizamos uma *framework* – ParJECoLi – que foi estendida usando AspectJ, uma linguagem baseada nos princípios da POA, para correr em ambientes de computação paralela. Depois da descrição da implementação (para ambas as abordagens), é feita uma comparação entre as implementações em Scala e em AspectJ. Para uma melhor comparação, sendo o ParJECoLi uma *framework*, é utilizado um caso de estudo para verificar o desempenho e comparar os resultados entre a implementação em Scala (ambas as abordagens usadas) e AspectJ.

7.1 DSLScala

Esta proposta, chamada *method proxy-based* pelos autores (Spiewak e Zhao, 2009) (traduzido: método baseado em proxy ou descrito de outra forma, talvez mais esclarecedora, delegação de invocação de método) consiste em usar Linguagem de Domínio Específico (LDE) – em inglês Domain Specific Language (DSL) - embebido na linguagem Scala para especificar *pointcuts* de um modo mais próximo de AspectJ. O seu uso permite especificar tipos de classes e assinatura de métodos, assim como permite o acesso a variáveis de contexto (Spiewak e Zhao, 2009).

Ao contrário do AspectJ, que possui um compilador que faz o *weaving* entre os aspectos e o código fonte (Gradecki et Lesiecki, 2003) (Laddad, 2012), neste trabalho recorreu-se ao uso de funções de ordem superior (*high-order*) – apresentadas no capítulo 5, secção 5.2.5 - para interceptar as chamadas a dados métodos e simular o funcionamento que o *weaving* permite. Apesar de dar uma sensação de uso mais próxima a AspectJ, acaba por ser invasiva face à ausência de um compilador próprio, o que leva a alterações do código original. Uma outra característica desta técnica é que não faz distinção entre os *jointpoints call* e *execution* (ver secção 2.1.12.1 para entender a diferença). É necessário recordar que se usarmos o *pointcut designator this, n*o primeiro caso o que vai acontecer é que o objecto a ser retornado é o objecto invocador, enquanto no segundo é o objecto invocado, ou seja, o possuidor do método chamado. Esta distinção é feita em AspectJ, mas torna-se quase impossível

implementá-lo usando técnicas do Scala dado que é a captura de um ponto de execução de grão fino, um pormenor muito específico. Para além desta razão, não existem ganhos significativos na implementação do ParJeCoLi, com a utilização de um ou do outro, ainda que um e outro sejam utilizados.

7.1.1 Estrutura

Na Figura 7. 1 - Implementação da classe Circle usando a framework proposta (Spiewak e Zhao, 2009). temos o exemplo da classe *Circle*, onde se pretende chamar o método *repaint* sempre que um dos selectores (*setters*), x_ ou y_ é chamado. De forma a evitar repetição de código, criando o fenómeno de *code tangling*, pretende-se então implementar um aspecto que chame *repaint* sempre que estes selectores são invocados, tal e qual como em AspectJ. No entanto, esta implementação é feita de forma diferente. Seguindo o exemplo da Figura 7. 1, vemos que no corpo do selector `x_(x:Int)` temos um método chamado defun e logo a seguir dentro de chavetas código. Este código é equivalente a: `defun((x:int) =>{ix=x})`, ou seja, o método `defun` é um método que recebe como argumento uma função. Isto é uma técnica, presente em Scala, que foi descrita na Secção 5.2.5, chamada funções ordem superior. A ideia é delegar a execução do código do selector aquele método. Ao delegarmos a execução do código, ao atribuir essa responsabilidade a outro, podemos desta forma acrescentar comportamento ou alterar o comportamento da mesma. Podemos, como o *advice before* presente em AspectJ, acrescentar comportamento antes, podemos, como o *advice after*, acrescentar comportamento depois de executar o código do selector (como é o caso deste exemplo, em que queremos chamar *repaint* depois de executado o código).

```
class Circle(...) extends AOP {
  ...
  /* weavable */
  def x_=(x:Int) = defun { ix = x  }

  /* not weavable */
  def y_=(y:Int) = { iy = y  }
}
```

Figura 7. 1 - Implementação da classe Circle usando a framework proposta (Spiewak e Zhao, 2009).

Uma pequena chamada de atenção para o facto de `defun` ser chamado no selector `x_` mas não no selector `y_`. Isto significa que o primeiro é *advisable* (Spiewak e Zhao, 2009), mas o segundo não, ou seja, é possível alterar o comportamento do primeiro mas não do segundo. Esta é uma característica desta proposta: para podermos acrescentar *advices*, ao estilo do AspectJ, é preciso delegar a execução do código ao método `defun`. Este pertence ao *Trait* AOP, conforme podemos ver na Figura 7.2, o que significa que qualquer classe que

implemente um método que queremos que seja *advisable*, precisa de estender este *Trait* (conforme podemos ver na Figura 7. 1, na classe Circle).

```
trait AOP {
  def defun[A](fun: =>A): A = {
    handleBefore()

    val back = try {
      fun
    } finally {
      handleAfter()
    }
    back
  }
}
```

Figura 7. 2 - Implementação do trait AOP.

Dados que as classes Java vêem *traits* como interfaces, foi necessário fazer algumas alterações na classe que implementa o *trait* (ver Figura 7. 3– neste exemplo, pretende-se delegar a `defun` a execução da função `run()`). Uma vez que vê como uma interface, significa que os métodos definidos não contam, ou seja, para a classe são vistos como métodos que têm de ser implementados na classe (ver Secção 5.2.2, do capítulo 5). Deste modo, foi necessário implementar o método `defun` na classe `AbstractAlgorithm`. Para contornar este obstáculo, foi criado um objecto (ver secção 5.2.6) com o mesmo nome e única coisa que o método tem de fazer é invocar esse objecto e chamar determinada função (algo que era feito pelo *Trait* AOP). Foi a forma encontrada para contornar esta limitação de interoperabilidade, embora retire alguma naturalidade à composição modular existente em Scala entre *traits* e classes, que permite ser menos invasiva (conforme comprovado na Figura 7. 1)

```
public abstract class AbstractAlgorithm<T
extends IRepresentation,S extends IConfiguration<T>> implements Serializable, IAlgorithm<T>, IDeepCopy, AOP {

      public IAlgorithmResult<T> run() throws InvalidConfigurationException, Exception {
          final AbstractAlgorithm bla = this;
          AbstractFunction0<IAlgorithmResult<T>> f = new AbstractFunction0<IAlgorithmResult<T>>() {
              @Override
              public IAlgorithmResult<T> apply() {
              //Código fonte da função run() original

              }
          };
          return this.defun(f);
      }

      public <A> A defun(Function0<A> fun) {
          return AOP$.MODULE$.methodIntercept(this,fun);
      }
}
```

Figura 7. 3 - Implementação do *Trait* AOP numa classe Java.

Uma outra chamada de atenção para o facto de serem passadas funções como argumento de um método e não métodos, uma vez que em Scala existe diferença entre um e outro (ver Secção 5.2.3, Capítulo 5). Uma vez que métodos definidos em classes Java não podem ser passados como argumentos, foi utilizada uma classe existente em Scala, chamada *AbstractFunction0*, a qual permite criar um objecto função que, por sua vez pode ser passado como argumento, conforme se pode ver no código da função `run()`, na Figura 7. 3.

```
class DistributedMigration extends Aspect{
    val teste = pointcut(classOf[jecoli.algorithm.components.algorithm.AbstractAlgorithm[_,_]]::('run)) -> *;
    val teste1 = pointcut(classOf[AddAspects]::('run)) -> *;

    before(teste) {
      println("Método achado e executado")
    }

    before(teste1){
      println("Método achado e executado")
    }
}
class AddAspects {

  def loadAspects(){
    println("Added!")
    AOP.addAspect(new DistributedMigration())
    println(AOP.aspects.length);
  }
}
```

Figura 7. 4 - Criação de um aspecto com pointcuts.

No final, podemos criar uma classe, a qual estende o *trait* `Aspect`, o qual vai permitir o programador usar a DSL para definir *pointcuts* e *advices*, conforme se pode ver na Figura 7. 4. O passo final é adicionar o aspecto a uma lista de aspectos, a qual é percorrida sempre que o método `defun` é chamado para verificar se existe algum *pointcut* sobre o método invocador e se sim, os *advices* daquele aspecto são executados.

7.1.2 Invasibilidade

Para permitir a captura de *joinpoints*, foi necessário alterar pontos no código fonte dado que, ainda que o Java e o Scala sejam compatíveis, existe alguma complexidade na sua interacção. Exemplo: o Java não possui conceito de função anónima ou função literal, algo que permite ao Scala passar o código fonte ou o corpo de um método como argumento de um outro método de forma directa. Esta complexidade na interacção entre ambas as linguagens levou a que fosse necessário realizar maiores alterações no código fonte, nomeadamente com a criação de um novo método e um maior número de linhas de código para interceptar aquele pretendido. Os exemplos a seguir ajudam a ter uma noção de como é uma técnica invasiva e por ser necessário interagir com Java, a sua invasibilidade é maior.

```
class Test1 extends AOP{
	def run(){
		defun{ println("bla bla bla bla")};
	}
}
```

Na classe em cima, o código do método é passado como argumento do método *defun*. Em Java, é necessário criar um objecto do tipo Scala *AbstractFunctionN* (sendo N o número de argumentos) e definir a função *apply* com o código fonte do método que queremos capturar. Para os casos em que o método não recebe argumentos, a implementação é mais simples (ver exemplo Java embaixo):

```
public class Test implements AOP{
	void run(){
	   AbstractFunction0<String> bla = new AbstractFunction0<String>(){

		@Override
		public String apply() {
			System.out.println("bla bla bla bla");
			return null;
		}
	   };
		defun(bla);
	}

	@Override
	public <A> A defun(Function0<A> fun) {
		return AOP$.MODULE$.methodIntercept(this,fun);
	}
}
```

Como se pode ver no exemplo, foi criada uma variável do tipo `AbstractFunction0` que representa funções que não recebem argumentos. O método *apply* não recebe argumentos, pelo que `AbstractFunction0` pode ser passado como argumento do método *defun*. No entanto, o caso muda de figura quando queremos passar funções que recebam argumentos. Em Scala, este problema não se nota, dado que providencia uma sintaxe que permite definir funções anónimas ou funções literais e o interpretador no momento da execução, substitui as variáveis no corpo pelos valores recebidos como argumento. Exemplo Scala embaixo:

```
class Test1 extends AOP{
	…

	def run2(a:Int, b:Int) = defun{println("o total é " + (a+b))}
}
```

A função run2 recebe dois argumentos, dois inteiros, e podemos ver no corpo do método que o seu código é passado como argumento de *defun*. O interessante é que as variáveis a e b são passadas como parte da função, e o compilador do Scala não dá nenhum aviso de erro. Isto porque o interpretador do Scala, quando a função for executada, consegue ir buscar os valores recebidos como argumento e substituir nas respectivas variáveis. Já em Java, tal não é possível, pelo que foi necessário recorrer a uma solução improvisada: foi criada uma classe FunctionN, que recebe no seu constructor a função e os argumentos necessários para ser executada.

```
class FunctionN[A,T](function:Function1[Seq[T],A], args:Array[T]){

	def getFunction = function

	def getArgs = args

}
```

É possível notar que o construtor recebe como argumento uma função do tipo *Function1*, mas esta função recebe como argumento uma sequência de Objectos, do tipo genérico. Apesar de o Scala possuir uma variedade de *Traits* (desde *Function0* até *Function22*) que representam funções, a verdade é que não existe um super tipo comum a estes *Traits*, ou seja, não é possível representar no construtor um tipo função genérico, que permita receber funções independentemente do número de argumentos. Como tal, improvisou-se esta solução para resolver o problema: receber uma função que recebe apenas um argumento, uma sequência – uma sequência de objectos - cujo tamanho pode variar consoante o número de argumentos que a função recebe. Para uniformizar, se quisermos passar uma função sem argumentos, simplesmente passamos uma sequência vazia.

De notar que sempre que queremos interceptar a chamada a um dado método, é necessário utilizar o método estático `methodIntercept` que está implementado no objecto AOP. Este método recebe como argumentos um objecto do tipo AOP, sendo este o objecto-alvo da invocação e um objecto do tipo FunctionN, que contém a função a executar e os argumentos. A classe que possui o método que queremos interceptar tem de implementar o *trait* AOP. No entanto, sendo uma classe Java, irá ser interpretado como uma interface. Só deste modo é que o objecto-alvo pode ser recebido como argumento do método estático.

Um outro problema encontrado, na interacção Java-Scala, e que levou a alterações no código fonte, foi o facto de o Java permitir *raw types* – não existir tipos parametrizados - enquanto que o Scala não permite isso. Isto levou a que por vezes fosse necessário definir no código Java esses tipos que não estão definidos.

Igualmente na definição do *pointcut designator* que permite obter o objecto invocador – isto para casos em que queremos que o *advice* obtenha o objecto invocador – torna-se necessário acrescentar uma linha de código no método que chama o método que se pretende capturar. Exemplo:

```
public IAlgorithmResult step() throws Exception,NonExistingNodeTypeException,
InvalidConfigurationException {
	CallingObject$.MODULE$.setCalling(this);
	algorithmResult = ga.run();
	return algorithmResult;
}
```

Dado que o Scala não tem nenhum mecanismo que permita passar um construtor como argumento de um método, delegando essa responsabilidade, para contornar esta limitação criou-se uma classe com métodos que permitem criar os objectos pretendidos. Em vez ser invocado o construtor para criar o objecto, a ideia é utilizar o objecto *Constructor* para criar o objecto pretendido. Isto acaba por ser mais uma medida que altera o código fonte.

7.1.3 Implementação

Com a descrição da invasibilidade desta técnica, interessa agora explicar com mais pormenor o funcionamento dela, desde que o método é interceptado até à execução do *advice* ou dos *advices* que estejam associados. Como foi dito em 7.1.2, secção que descreve a invasibilidade, um método é interceptado usando o método estático do objecto AOP. Este recebe como argumento o objecto invocado, assim como um objecto FunctionN, o qual possui o código da função a executar e um *array* com os argumentos. O código do método que queremos executar está encapsulado no objecto FunctionN. Se o código fonte estivesse em Scala, seria diferente.

O funcionamento desta técnica depende de obter a `StackTrace` da execução. Uma vez que não possuímos um compilador próprio que possa criar pontos de junção e associar a determinados *advices*, torna-se necessário recorrer a uma alternativa para descobrir qual o método que foi interceptado para que se possa executar os *advices* sobre esse ponto do código.

7.1.4 Aspectos

Dentro do projecto que foi criado, os aspectos são classes Scala, que possuem membros e métodos. Possuem *pointcuts* que são guardados em valores e não em variáveis, para garantir que os mesmos não são alterados nunca durante a execução e garantir o funcionamento dos *advices* associados a eles. Estes são nada menos que métodos, que recebem como argumento um *pointcut.* A forma de funcionamento é explicada mais à frente. Normalmente, na maioria dos casos implementados no ParJecoli, dado que os aspectos possuíam membros ou métodos estáticos, a classe possui um objecto companheiro onde estão definidos estes.

7.1.5 Pointcuts e Advices

São essenciais para que possamos adicionar código antes ou depois de um método ser interceptado ou contornar a sua execução. Quando definimos um *pointcut*, pode-se associar *advices* ao mesmo:

```
class DistributedMigration extends Aspect with AbstractMigration{

	(…)

	//pointcuts...
	val algorithmRun =
pointcut(classOf[jecoli.algorithm.components.algorithm.AbstractAlgorithm[IRepr
esentation,IConfiguration[IRepresentation]]]::('run)) -> *;

	after(algorithmRun){
		(c:AnyRef, args:Array[Object]) =>{
			aux.synchronized{
				DistributedMigration.pool.shutdownNow()
			}
		}
	}
}
```

Existem 3 tipos de *advices*, o que significa que existem 3 métodos diferentes em que cada um dos métodos representa um *advice*: **before**, **around** e **after**. Existem diferenças entre eles, nomeadamente no número e tipo de objectos a que têm acesso. O *advice before* tem acesso ao objecto alvo de invocação e aos argumentos que o método interceptado vai receber. Alterando os valores dos argumentos vai alterar, automaticamente, o resultado da execução. Isto acontece porque o objecto a que tem acesso é a mesma instância que irá ser utilizada quando o método for executado. O *advice around* tem acesso aos mesmos objectos que o anterior, podendo alterar à mesma os argumentos mas existe uma diferença: quando é chamado, o programador tem a possibilidade ou não de indicar se quer executar o código do método interceptado. Existem duas hipóteses: ou executa o código chamando o método

```
def proceedWith[A,T](args:Seq[T]) = {
     if(aroundPrivledge){
          var y =
FunctionN.convert(function.asInstanceOf[FunctionN[A,T]].getArgs)
          var fun = function.asInstanceOf[FunctionN[A,T]].getFunction
          try {
               back = Some(fun(y))
            } finally {
```

```
                executeAfterAdvices
            }
        }
    }
```

O que este método faz é prosseguir a com a execução do método, tal e qual como o *proceed()* do AspectJ. De notar que este método que recebe como argumentos, um *array* de objectos. Isto acontece para o caso de se pretender alterar os argumentos da função, alterando assim o resultado da execução. Um outro ponto importante de referir nesta função é que no fim é chamada uma função para executar os *after advices*.

Em relação ao *pointcuts*, há um detalhe que é necessário mencionar, e que, de certa forma, torna a classe, que emular o funcionamento de um aspecto, fortemente acoplada com a classe no código fonte: é necessário especificar a classe do método que se pretende interceptar no *pointcut*. Isto acaba por ser algo limitativo.

7.1.6 Interceptar um método comum

Quando um método é interceptado, o código que está no seu corpo é passado como argumento para um método dentro uma outra entidade, um objecto chamado AOP. Essa entidade recebe o código do método a executar mas não recebe a assinatura do método, o que significa que não sabe a que método aquele código pertence. Isto significa que temos de recorrer à *StackTrace* (contém informação sobre os métodos invocados que ainda não retornaram valor ou acabaram a execução) do programa de modo a descobrir qual poderá ter sido o método que foi interceptado. É claro que a pilha pode conter nomes de vários métodos, uma vez que métodos invocados dentro de métodos são empilhados, sendo que o primeiro elemento da pilha é o método mais recentemente invocado e o último o mais antigo. Logicamente, o mais antigo será sempre o método *main*, uma vez que é o primeiro método a ser invocado dentro do programa.

Uma vez invocado o `methodIntercept`, vai criar uma instância da classe AOPClass, vai guardar uma cópia da pilha, obtida naquele momento, vai definir que a pilha que será usada pela instância AOPClass será a pilha obtida e depois vai realizar outros passos que são importantes e serão explicados o porquê mais à frente. No fim, invoca um método dessa instância com o mesmo nome: `methodIntercept`. E é então nesse método que que é verificado qual o nome do método que foi interceptado, verificando a pilha, e, uma vez obtido o nome, podemos então começar a verificar se existem *advices* que se pretendem executar. A execução dos *advices* é semelhante à do AspectJ, é feita por aspecto. Desta forma, é importante a ordem de introdução dos aspectos na lista, sendo que a ordem de execução é FIFO. A execução dos *advices before* vem em primeiro lugar, dado que o seu objectivo de utilização é que sejam executados antes da função. A execução dos *advices after* é o contrário, pretende-se que sejam executados depois da função. Estes dois tipos são casos

simples de resolver. No entanto, existe um outro tipo de *advice – around-* que foi mais difícil de emular e isto devido a várias razões. O objectivo do *advice around* pode nem sempre ser o mesmo, dependendo da forma com que se pretende utilizar. Como o nome indica, passa à volta daquele ponto de execução, e, deste modo, de qualquer método que se encontre naquele ponto de execução. É possível ao programador determinar se quer prosseguir com aquele ponto de execução ou não. No entanto, esta última opção vai influenciar automaticamente a execução dos restantes aspectos e a razão é simples: se não se pretender que aquele ponto seja executado pode deixar de fazer sentido a execução de passos antes ou depois. Se se pretende que a função seja executada, basta usar o método `proceedNoReturn()` (ver exemplo embaixo):

```
around(algorithmRun){
	(algorithm:tests.Hello, args:Array[Object]) =>{
		println("TestAspect2 - around before")
		AOP.getCurrentTier.proceedNoReturn
		println("TestAspect2 - around after")
	}
}
```

O que se pretende com a invocação deste método é prosseguir com a execução do programa naquele ponto e, possivelmente, com a execução do método. Existem, no entanto, dois outros modos de prosseguir: no caso em que se pretenda continuar com a execução do método mas com argumentos diferentes, alterando assim o resultado final e o caso em que se pretenda continuar com a execução do programa mas com um resultado – os métodos `proceedWith` e `proceedWithResult`. Em ambos os casos queremos que o programa prossiga com aquele ponto de execução, ou seja, não foi esquecido mas pretende-se que continue com um resultado diferente daquele que seria o natural. Uma consequência da invocação deste método é que os restantes aspectos com *advices* sobre aquele ponto de execução também vão ser executados. De certa forma, o *advice around* dá ao programador não só o poder de decidir se pretende que aquele ponto no código seja executado, como também influenciar o seu resultado final e ainda se outros aspectos serão executados ou não. Este último ponto faz sentido, dado que se não se pretende que aquele ponto no código fonte seja executado, logo talvez não faça sentido que *advices* sejam executados (com a excepção de *advices* de aspectos que venham antes deste).

A outra possibilidade é a de não prosseguir com a execução do método interceptado. Neste caso, o que acontece é que os aspectos a seguir não vão ser executados, como foi dito, por estarem ligados aquele ponto de execução – e se não há intenção de prosseguir com a sua execução, deixa de fazer sentido. Existe o caso, no entanto, de aquele método exigir o retorno de um valor, caso contrário toda a execução irá dar excepção e o seu funcionamento fica comprometido. Sendo assim, para métodos que não retornem valor *void* torna-se necessário utilizar o método `returnResult`.

Foi dito que possivelmente se iria prosseguir com a execução do método interceptado (ou com argumentos diferentes ou com um resultado específico) mas tal pode não acontecer. A explicação é simples: pode-se dar o caso de haver mais um aspecto que possua igualmente um *around advice*. E pode-se dar o caso de não prosseguir com a execução. Se tal acontecer e for necessário retornar um valor, aquele que este *advice* retornar é o definitivo.

O comportamento dos *advices* aqui descrito está de acordo com o comportamento dos mesmos no AspectJ. Para o caso do ParJeCoLi, no entanto, não é utilizado a fundo todas as capacidades ou existe algum caso de "emparelhamento" de *around advices*.

Um problema desta técnica, que foi detectado e solucionado, reside no facto de vários métodos interceptados poderem usar a mesma *stack trace* – a qual nos permite obter a sequência de métodos chamados, numa pilha. No exemplo alvo usado, o ParJeCoLi, existe o caso de num dos aspectos (*traits*) haver um *pointcut* que pretende capturar a chamada a um método dentro de um outro aspecto. Seguindo a linha de execução seria: um método é interceptado pelo *trait* A, possivelmente o *advice* associado a esse *pointcut* chama um método dentro desse mesmo *trait*, o qual é interceptado por um *pointcut* definido dentro do *trait* B. O possível problema dentro deste caso, com esta técnica, seria que a pilha a ser usada em ambas as intercepções é quase a mesma (com mais alguns métodos do próprio código de intercepção e execução de *advices*). Para solucionar este problema, foi necessário que cada método interceptado exclua da sua pilha métodos que estão presentes na pilha de um método que foi interceptado anteriormente, ou seja, no fundo a pilha que se obtém é a diferença entre a pilha anterior e a pilha actual. O resultado é que ficam os métodos que não foram invocados antes (há métodos que podem ser invocados outra vez). Desta forma, temos a certeza que não vamos tornar a interceptar métodos que já foram interceptados, fazendo com que o programa entre num ciclo infinito.

7.1.7 Interceptar o método main

Para o caso em que se pretende interceptar a execução do método *main*, é necessário recorrer a uma estratégia que não é invasiva, mas vai trazer alguma complexidade. A ideia consiste em chamar, dentro de um objecto, o método *main* que se pretende executar. O exemplo embaixo explica esta ideia de forma fácil:

```
object AOPMode extends AOP{
       def main(args: Array[String]) {
              var add = new AddAspects();
              add.loadAspects();
              var create = (args:Seq[String]) => {
              CountOnesCAGATest.main(args.toArray)
      }
      AOP.methodInterceptMain(this, new FunctionN(create, args))
```

```
    }
}
```

O objecto `AOPMode` possui o seu próprio método *main*, no qual é chamado, no exemplo utilizado, o teste `CountOnes`. Ele é interceptado pelo método `methodInterceptMain`, que recebe os mesmos argumentos que o método `methodIntercept` mas possui uma execução diferente desta.

7.1.8 Construtores

Foi referido na secção anterior que não era possível interceptar a chamada a um construtor utilizando a mesma técnica para interceptar chamadas a métodos, dado que o Scala não permite passar aquele como um argumento. Para ser possível então interceptar a chamada a um construtor, foi criada uma classe com métodos, cada um permitindo criar o objecto que se pretende, sendo desta forma interceptáveis. Esta classe está definida em linguagem Scala, uma vez que se pretende utilizar a técnica de *mixins* para poder criar declarações inter-tipo, assim como para executar *advices* sobre essa parte do código.

Como foi referido, a utilização de uma classe com métodos responsáveis por construir uma instância de uma classe, permite que sejam executados *advices* sobre eles. A ideia é "transformar" os construtores em métodos para serem interceptados, como se fossem métodos comuns. Recorrendo a um exemplo torna-se mais fácil de explicar:

```
class Constructor extends AOP{

  def createCellularGeneticAlgorithm(config:CellularGeneticAlgorithmConfiguration
[ILinearRepresentation[java.lang.Boolean],AbstractLinearRepresentationFactory[java.lang
.Boolean]]) = {
        type U =
CellularGeneticAlgorithmConfiguration[ILinearRepresentation[java.lang.Boolean],AbstractLinearRepre
sentationFactory[java.lang.Boolean]]
        var t = (f:Seq[U]) =>{
          var b = f.toArray;
          new CellularGeneticAlgorithm with IAlgorithmExtension;
        }
        var args = Array[U](config)
        AOP.methodIntercept(this, new FunctionN(t, args))
  }


  def createMultiGA
(mgp:MultiGAPars,topol:GATopology,algorithms:ArrayList[IAlgorithm[_]]):MultiGA={

    var create = (args:Seq[Object]) => {
      new MultiGA(args(0).asInstanceOf[MultiGAPars],args(1).asInstanceOf[GATopology],
          args(2).asInstanceOf[ArrayList[IAlgorithm[_]]])
    }
    var args = Array[Object](convert(mgp),convert(topol),convert(algorithms))
       return AOP.methodIntercept(this, new FunctionN(create, args))
```

```
  }

}

object Constructor {
  var construct = new Constructor

  def getConstruct():Constructor = construct
}
```

O objecto `Constructor` possui uma instância da classe `Constructor`. Quando queremos criar um objecto, é necessário aceder ao objecto e utilizar a instância que possui para invocar o método pretendido. Podemos ver no exemplo que é chamado o método `methodIntercept` do objecto AOP em qualquer um dos métodos. Ao método são passados a instância da classe, e um outro objecto do tipo `FunctionN`, com uma função cuja única finalidade é executar o construtor da classe para criar o objecto pretendido e com os argumentos necessários para esse construtor. Se existir nalgum aspecto algum *pointcut* associado a qualquer um dos métodos presentes em `Constructor`, são verificados então os *advices* que lhes estão associados. Podemos embaixo um exemplo de um *pointcut* que pretende capturar a execução e um *advice* com código a executar (antes ou depois):

```
class DistributedMigration extends Aspect with AbstractMigration{

        (…)
        //pointcuts...
        val gatopology=pointcut(classOf[com.codecommit.aop.Constructor]::('createGATopology)) ->
classOf[GATopology];
        val multiga=pointcut(classOf[com.codecommit.aop.Constructor]::('createMultiGA)) ->
classOf[MultiGA];

        after(gatopology){
        (c:AnyRef, args:Array[Object]) =>{
            var GAtopology = c match {
                    case c1: GATopology => c1
                    case _ => throw new ClassCastException
           }
            gr = GAtopology.getGraph
          }
        }

        after(multiga){
          (c:AnyRef, args:Array[Object]) =>{
            var m = c match {
                    case c1: MultiGA => c1
                    case _ => throw new ClassCastException
            }
            var gas = m.getPops()
            for(i <- 0 to gas.size()){
               var gaExtension = gas.get(i).getGa() match {
                       case c1: IAlgorithmExtension => c1
                       case _ => throw new ClassCastException
               }
               gaExtension.setDestinations(gr.getDestinations(i));
               gaExtension.setArrivals(gr.getArrivals(i));
               gaExtension.setIterationStep(itStep);
```

```
                gaExtension.setMigrants(AbstractMigration.numMigrants);
                gaExtension.setSync(syncronized);
            }
          }
        }

}
```

Dito de forma simples, a ideia foi "transformar" os construtores em métodos. Os construtores são encapsulados numa função, são passados como argumento para um outro método, e segue o processo como se de um método regular se tratasse.

Para explicar com mais pormenor como é criado uma instância usando o objecto e classe Constructor, temos um exemplo em que queremos criar um objecto do tipo *CellularGeneticAlgorithm*:

```
public class CountOnesCAGATest {


/**
 * The main method.
 *
 * @param args the arguments
 */
public static void main(String[] args)
{
       try {

              (…)

                     IAlgorithm<ILinearRepresentation<Boolean>> algorithm =
              (IAlgorithm<ILinearRepresentation<Boolean>>)

                     Constructor.getConstruct().createCellularGeneticAlgorithm(configu
              ration);

              algorithm.run();


       } catch (InvalidNumberOfIterationsException e) {
              e.printStackTrace();
       }
       catch (InvalidConfigurationException e) {
              e.printStackTrace();
       }
       catch (InvalidSelectionParameterException e) {
              e.printStackTrace();
       } catch (Exception e) {
              e.printStackTrace();
       }
}
}
```

O processo que se segue a seguir é o mesmo que se dá quando se intercepta métodos comuns.

7.1.9 Declarações Inter-tipos

Existe uma diferença entre as implementações: enquanto que AspectJ permite utilizar a declaração inter-tipo para acrescentar métodos e variáveis a uma interface e, desta forma, todas as classes que implementem aquela interface possuem-nos, em Scala não é possível fazer o mesmo, da mesma forma. A extensão da interface, recorrendo aos inter-tipos (ver secção 2.1.3) é feita de forma estática mas indirecta, ou seja, a estrutura da interface não é alterada no código fonte. Esta extensão é feita durante o tempo de compilação no processo que é conhecido em AspectJ como *weaving* (ver secção 2.1.2), que é um mecanismo próprio do compilador utilizado pela linguagem,e permite que o código fonte daquela seja alterado mas não directamente.

Em Scala não existe forma de fazer isto, e, mesmo recorrendo às técnicas existentes, mais difícil se torna com a interacção com Java. A técnica que foi utilizada é chamada *mixin*, a qual pode ser utilizada estaticamente, em que a estrutura da interface seria alterada no código fonte, ou dinamicamente, na criação de uma instância de uma dada classe que implemente aquela interface. O problema da primeira abordagem é que, para além de que é invasiva na estrutura da interface, o Java não reconhece *traits* como *traits*, mas sim como interfaces, o que significa que não vai reconhecer membros e que métodos que estejam definidos não vão ser considerados como tal, mas sim como declarações. Tal vai obrigar as classes implementadoras da interface a implementar esses mesmos métodos, aumentando a invasibilidade grandemente. Esta limitação deita por terra toda a potencialidade de utilizá-los estaticamente para emular inter-tipos. Desta forma, a solução encontrada foi dentro dos *mixins* dinâmicos, ou seja, no momento em que a instância é criada, realizar o *mixin*. O problema (mais um) é que não é possível realizá-los dentro das classes Java, uma vez que não são reconhecidos pelo compilador do mesmo, conforme foi dito na secção anterior. A solução passou então por "delegar" essa responsabilidade para uma classe Scala, com métodos que criam as instâncias pretendidas e devolvem-nas, realizando o *mixin* com o *trait* que vai funcionar como inter-tipo.

No último exemplo, no método `createCellularGeneticAlgorithm`, podemos ver que é acrescentado o *trait* `IAlgorithmExtension` que estende instâncias que implementem a interface `IAlgorithm`, como é o caso do tipo `CellularGeneticAlgorithm`. Pelo exemplo acima, podemos ver que usamos *mixins* dinâmicos para emular o funcionamento dos inter-tipos, com uma grande diferença: não estendem a interface, como é feito em AspectJ, estendem sim instâncias de classes implementadoras da interface `IAlgorithm`. Esta abordagem possui vantagens e desvantagens: a forma como é implementada em AspectJ é muito mais fácil uma vez que segue uma aproximação top-bottom, ou seja, as alterações são feitas na interface e automaticamente qualquer classe implementadora herda as capacidades

dadas. Uma das características é que todas as instâncias vão herdar membros e métodos do inter-tipo e, ao mesmo tempo, vão manter o seu próprio estado dessas variáveis. Por exemplo, se fosse acrescentado uma variável inteiro que servisse como contador, cada instância podia ter um estado diferente em relação a essa variável. Uma desvantagem talvez desta abordagem dar-se-ia no caso em que pretendêssemos que apenas algumas instâncias de classes tivessem aquelas variáveis e métodos. Claro que seria fácil fazer tal, se se acrescentasse uma interface comum apenas para certas classes, ou usando inter-tipos directamente sobre as classes. No entanto, se quiséssemos características diferentes em duas instâncias da mesma classe (uma instância com e outra sem), já não seria possível. Na forma como os inter-tipos são implementados neste trabalho, é mais fácil gerir caso a caso embora dando mais trabalho no caso em que não se pretende isso.

Interessa referir que se estivéssemos a lidar somente com a linguagem Scala, seria possível implementar *mixins* de forma estática. Isto significa que caso estivéssemos interessados em implementar um inter-tipo do género top-bottom – estender um *trait* que seja implementado por várias classes, permitindo assim que elas também fossem estendidas - tal seria possível, embora levasse a uma alteração na estrutura do *trait* e, como consequência, do código fonte. Não foi possível, no entanto, até à data, encontrar uma técnica em Scala que permitisse alterar a estrutura de um *trait* externamente. Tendo em conta que se pretende emular o funcionamento do AspectJ em Scala, sabendo que é necessário haver invasão do código fonte para implementar os seus mecanismos, de certa forma é aceitável esta alteração de estrutura. Como foi dito, é possível fazer o mesmo, não da mesma forma.

7.1.10 Avaliação

Esta técnica acaba por alterar uma parte do código fonte – necessário para que a responsabilidade de executar uma função seja delegada através do uso de funções ordem superior. Esta alteração é maior ainda quando se utiliza classes Java. Como tal, não cumpre totalmente o princípio de *obliviousness* presente em AspectJ (Spiewak e Zhao, 2009), embora tal não seja muito importante, uma vez que neste trabalho o nosso foco seja mais a modularização de aspectos. Sendo os aspectos implementados por *traits*, uma unidade de modularização pura, será interessante ver qual o impacto a nível de composição modular num sistema grande.

Uma das limitações desta técnica é o facto de não poder indicar no *pointcut* o tipo de argumentos dos métodos que queremos capturar. Se tivermos perante uma classe com *overloading* de métodos, esta técnica não vai distinguir entre métodos com o mesmo nome, ainda que a sua assinatura seja a mesma. Para o exemplo utilizado, o **ParJeCoLi**, não é relevante porque não existe *overloading* nas classes sobre os quais os aspectos agem. No caso de termos *overloading* de construtores, não há este problema uma vez que é usada uma

classe especial para criar objectos, isto quando se pretende interceptar a sua chamada. Como tal, é possível especificar qual o construtor que se pretende capturar, especificando o nome do método. Nada impede, assim, de criar métodos com nomes diferentes para quando o construtor receber argumentos diferentes ou diferente número de argumentos. Esta limitação deve-se ao facto de a pilha devolver o nome dos métodos que são invocados, assim como a que classe pertencem, ficheiro e ainda até ao pormenor da linha nesse mesmo ficheiro. No entanto, não dá indicação de quais ou quantos argumentos são chamados, pelo que se torna impossível entender com certeza qual a assinatura do método invocado.

Igualmente confirma-se que é uma técnica que obriga a alterações no código fonte, principalmente alterações necessárias para interceptar que acabam por ser bastantes. O facto de não ser um sistema apenas em Scala, impede de utilizar vários mecanismos presentes no Scala que diminuiriam o número de linhas de código acrescentadas e mesmo as alterações no código fonte seriam mais discretas. Esta técnica seguramente seria menos invasiva e mais discreta se a aplicação alvo fosse programada em Scala. Para além disso, existem alguns problemas em relação aos tipos parametrizáveis de uma classe. Em Java pode-se usar os chamados *raw types*, uma técnica que permite declarar tipos genéricos sem especificar que tipos lhe vão ser atribuídos como parâmetros. Em Scala não é possível usar esta técnica, ou seja, é necessário especificar os parâmetros, podendo-se usar genéricos. O problema é que por vezes o código fonte do Java pode querer passar um tipo que recebe parâmetros mas que não estão especificados, ou seja, o tipo foi criado mas sem parâmetros especificados e, como tal, quando o argumento é recebido por um método (ou função) em código Scala, o compilador vai declarar a existência de um erro. Logicamente obriga a que sejam especificados. Uma outra alternativa é o uso de *wildcards* no lugar dos parâmetros, mas esta solução nem sempre funciona.

Uma outra limitação é o facto de não ser possível estabelecer uma ordem de precedências entre *advices* de aspectos, até mesmo entre aspectos diferentes. É possível estabelecer a ordem dos aspectos, sendo que os primeiros serão executados primeiro e se quisermos ser mais específicos – por exemplo, num dado caso interessa-nos que o *advice before* do aspecto A seja executado primeiro do que o *advice before* do aspecto B - a ordem dos aspectos na lista até resolve este problema. Mas, se tivermos mais um caso específico – queremos que o *advice after* do aspecto B seja executado primeiro do que o do aspecto A, tendo em conta o caso anterior já não é possível.

7.2 Scala Mixins

Esta proposta utiliza uma unidade de modularização do Scala, cujo principal ponto forte é a reutilização – o mecanismo de *traits* – e uma técnica baseada na sua utilização – os *mixins* (ou mistura – ver 5.2.1). Esta solução foi apresentada por (Bonér, 2008), o qual chama “Scala

Mixins" e é igualmente mencionada em (Spiewak e Zhao, 2009) como uma possível solução para emular AspectJ.

7.2.1 Estrutura

Basicamente, a ideia na sua essência é utilizar a técnica de *mixins*, baseando-se na utilização de *trait* – recorde-se que é um módulo desenvolvido ou introduzido em Scala para permitir reutilização de código – para alterar o comportamento de funcionalidades ou adicionar estado e comportamento. Isto é possível dado que os *traits* permitem definir métodos, os quais podem alterar/acrescentar comportamento a um método numa outra classe (em inglês chama-se *stackable modification using scala mixins*).

A Figura 7.5 apresenta a estrutura desta proposta e na Figura 7. 6, Figura 7. 7, Figura 7. 8 e Figura 7. 9 podemos ver um exemplo baseado naquele mostrado em (Bonér, 2008) mas adaptado para a nova versão de Scala.

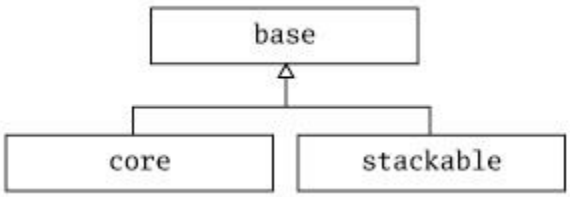

Figura 7. 5 - Estrutura de scala mixins para permitir alteração de comportamento (Venners, 2009).

O elemento `base` (exemplo Figura 7. 6) pode ser um *trait* ou uma classe (abstracta) e o seu papel é, tal como numa interface de serviços (Bonér, 2008), apresentar os métodos que queremos alterar o comportamento. Estes métodos vão ser implementados pelo *Core* (Figura 7. 7) – uma classe concreta – e pelo `stackable` (Figura 7. 8) que no fundo vai usar o sistema de *mixin* para introduzir novo comportamento aos métodos implementados naquele.

```
trait Stuff {
    def doStuff
}
```

Figura 7. 6 - Trait que representa o elemento base com o método que se pretende modificar.

```
class RealStuff extends Stuff{
    def doStuff = println("doing real stuff")
}
```

Figura 7. 7 - Classe RealStuff é o elemento Core.

```
trait LoggableStuff extends Stuff {
  abstract override def doStuff {
    println("logging enter")
    super.doStuff
    println("logging exit")
  }
}
```

Figura 7. 8 - Trait que modifica o comportamento do método da classe RealStuff.

No exemplo demonstrado, podemos ver como podemos utilizar o scala *mixins* para emular os *advice before* e *after*. O *trait* `LoggableStuff` faz `override` do método e adiciona comportamento antes de chamar o método `doStuff` presente na classe `RealStuff` – o elemento *Core* – e depois. Este comportamento é por si só equivalente ao *advice around*, do AspectJ.

----------Exemplo sem mixins----------

```
scala> val stuff = new RealStuff

stuff: stacks.RealStuff = $anon$1@6732d42
scala> stuff.doStuff

doing real stuff
```

----------Exemplo com mixins----------

```
scala> val stuff2 = new RealStuff with LoggableStuff
stuff2: stacks.RealStuff with stacks.LoggableStuff = $anon$1@1082d45

scala> stuff2.doStuff
logging enter
doing real stuff
logging exit
```

Figura 7. 9 - Resultado antes e depois de utilizar mixins.

A Figura 7. 9 mostra simplesmente o resultado da utilização de *mixins* – o comportamento original é modificado pela mistura com o `Trait LoggableStuff`.

7.2.2 Invasibilidade

Pelo padrão apresentado, torna-se claro que é necessário que tanto *Core* como *Stackable* tenham uma interface em comum. Se se der o caso de a classe em *Core* não implementar uma interface, é necessário acrescentar à sua declaração uma interface que será também implementada pelo *Stackable*, embora noutras condições, o que se torna um caso claro de

invasibilidade ao código fonte. No entanto, podemos declarar nessa interface todos os métodos aos quais pretendemos "capturar" o comportamento, através dos *mixins*. Um ponto a favor é que essa interface pode declarar três métodos diferentes, por exemplo, e nós só querermos implementar um aspecto que capture um desses métodos. Não somos obrigados a implementar os restantes. Isto acontece porque para implementar os aspectos utilizamos *traits* e não classes, e como tal, estes não são obrigados a implementar todos os métodos, mas é-lhes permitido fazer *overriding* e assim alterar comportamento – contanto que os métodos da interface comum estejam implementados na classe *Core*, algo que alias o Java faz questão de que aconteça, caso contrário não compila o código. A vantagem desta aproximação é que podemos implementar aspectos que façam *override* apenas dos métodos pretendidos, e não de todos, evitando assim que a classe *Core* implemente várias interfaces, tornando-se mais invasivo. Se tal não fosse possível, seríamos obrigados a partir a interface em duas, ambas com métodos que aquela implementa, mas apenas uma teria os métodos que o aspecto implementa. Se um outro aspecto capturar métodos diferentes, seria necessário partir aquelas interfaces em três ou quatro - todas as interfaces teriam métodos implementados pela classe *Core*, mas uma teria métodos que não interessavam aos aspectos, uma teria aqueles que interessava a um aspecto, uma outra para o outro aspecto e, possivelmente, uma quarta com métodos que interessasse capturar a ambos os aspectos. Portanto, de uma interface passamos, possivelmente, para quatro, e apenas com dois aspectos! A possibilidade de fazermos *overriding* é uma vantagem que simplifica a implementação.

Implementando apenas os métodos que queremos, mantém o código em cada aspecto limpo, assim como na classe *Core* e direccionando-os apenas para o que se pretende capturar. Na implementação do ParJeCoLi foi necessário criar interfaces – *IslandInterface*, *IMultiGA*, *IMain* – para se poder capturar métodos.

Existe também o caso em que os métodos que uma classe implementa estão declarados já numa interface, podendo assim o aspecto utilizar a interface já existente – mas fazendo apenas *override* dos métodos que lhe interessam.

Um caso também à parte prende-se com a intercepção de um método abstracto. Na ferramenta que foi estendida, pretende-se capturar o método *iteration* presente na classe abstracta *AbstractAlgorithm*. O problema é que este método foi declarado como *protected*, ou seja, só as classes que estendam a classe abstracta é que podem ter acesso a ele, não estando portanto declarado em nenhuma das interfaces implementadas. A forma de contornar este problema é utilizar a própria classe abstracta como *Base*, ou seja, em vez de partilharem uma interface comum, ambas estendem a mesma classe. É certo que qualquer classe *Core* implementa aquele método abstracto, logo é só uma questão de estender no aspecto. Um problema que pode surgir é se tivermos um aspecto que já estende uma outra classe. Para resolver este problema, surgem duas opções, dependendo cada uma do que se quer. A primeira solução consiste em dividir o aspecto, ou seja, criar um outro apenas dedicado a capturar métodos de uma mesma classe *Core*, enquanto que o original trata dos restantes.

Esta solução apresenta-se provavelmente como a mais desejável, dado que não é mais invasiva. Uma outra hipótese consiste em declarar esse método na interface. No entanto, o efeito desta alteração é que se o método for privado ou protegido, deixa de o ser, alterando de uma forma mais profunda não só a classe *Core* mas também o funcionamento de toda a estrutura. Por este motivo, torna-se menos desejável, embora é possível, tendo o conhecimento do funcionamento do código fonte, se vai ter repercussões indesejáveis. Se com efeito não temos esse conhecimento, o melhor é optar pela primeira opção.

Pode haver casos em que um aspecto implementa interfaces que possuem métodos com o mesmo nome, o que causa conflitos. Neste caso, uma das soluções passa por pegar numa das interfaces e dividi-la em duas interfaces, uma com os métodos que aspectos pretendem capturar e outra com os restantes métodos. Esta última vai estender a primeira e mantém-se a interface que a classe *Core* implementa. Sempre que houver um método da classe *Core* que um aspecto pretenda capturar, torna-se necessário modificar as interfaces. No exemplo implementado, o ParJeColi, existe um conflito com o método *deepCopy()*: ambas as interfaces *IAlgorithm* e *IEvaluationFunction* declaram um método com o mesmo nome, embora com valor de retorno diferente. Uma outra solução poderá ser a de particionar, mais uma vez, o aspecto em dois, ficando com uma parte mais focada naquela classe *Core* e a parte original no restante. Desta forma, evitam-se alterações no código fonte.

7.2.3 Implementação

Uma vez descrita a invasibilidade da técnica, interessa agora descrever a forma como emula o funcionamento de alguns dos mecanismos do AspectJ. A ideia base é utilizar uma interface comum entre classe presente no código fonte e o aspecto, para interceptar a chamada a um dado método e, desta forma, modificar o seu comportamento. Foi visto que esta interface comum pode já existir ou não, e pode ser necessário adaptar-se para casos específicos (tudo descrito na secção anterior). Até podemos nem usar uma interface comum mas uma classe abstracta, conhecendo claro algumas das limitações que acarreta. Embaixo podemos ver um exemplo simples, usando o ParJeCoLi como código fonte:

```
trait DistributedMigAlgorithm[T <: IRepresentation] extends IAlgorithm[T]{
   //before e after - método run, abstractAlgorithm
      abstract override def run:IAlgorithmResult[T] = {
      var pcd = new Within("Island", true)
      var extension = this match{
            case c:IAlgorithmExtension => c
      }
      if(pcd.checkWithin){
            AbstractMigration.itStep = extension.getIterationStep
            AbstractMigration.numMigrants = extension.getMigrants
            DistributedMigration.syncronized = extension.getSync
```

```
        }
        var result = super.run
        DistributedMigration.aux.synchronized{
               DistributedMigration.pool.shutdownNow()
        }
        return result
        }
  }
```

Em ScalaMixins, a ordem de precedência é a ordem de mistura com a classe *core*, ou seja, a ordem em que os *traits* são misturados vai definir a ordem de qual vai ser o primeiro a executar. De notar que é possível um *trait* evitar a execução de outros ou mesmo do método da classe core, bastando para isso não chamar a palavra `super.` Se o método retornar algum valor, é necessário que o método no *trait* evita a execução retorne algum valor, caso contrário dá erro na execução.

7.2.4 Construtores

Nos ScalaMixins também foi necessário implementar os construtores da mesma forma como foram implementados em DSLScala. Basicamente foi criada uma classe, chamada `Constructor`, juntamente com um objecto companheiro com o mesmo nome, que permitem criar instâncias de classes que se pretende capturar o construtor. Noutros casos pretende-se criar um *advice* sobre o construtor e, como tal, é feito um *mixin* entre o aspecto e esta classe, noutros casos pretende-se apenas criar um inter-tipo e, como tal, é necessário fazer um *mixin* no momento em que o objecto é criado. Também é utilizada para fazer o *mixin* entre o aspecto e a classe *Core*.

7.2.5 Declarações Inter-tipos

A solução encontrada para implementar Inter-tipos foi a mesma utilizada na solução DSLScala (ver secção 7.1.9).

7.2.6 Método Main

Para capturar a chamada ao método *main*, a ideia utilizada foi a de criar uma classe com um método que chamasse estaticamente o método main da classe que se pretende. A ideia é baseada na que foi utilizada na técnica anterior, mas em vez de delegar a execução do código, a ideia é utilizar os *mixins*. O exemplo demonstra claramente a ideia:

```
object AOPMode {
       def main(args: Array[String]) {
          val p = new AOPModeClass
          p.main(args)
       }
}

trait IMain {
       def main(args:Seq[String]) = {
         CountOnesCAGATest.main(args.toArray)
       }
}

class AOPModeClass extends IMain{
       override def main(args:Seq[String]) = {
         CountOnesCAGATest.main(args.toArray)
       }
}
```

O objecto AOPMode é chamado para correr a aplicação. Ele cria um objecto AOModeClass e depois usa-o para invocar o método main. Este método é o que vai ser interceptado pelo aspecto. Para isso, criou-se um *trait* que vai funcionar como interface comum entre a classe e o aspecto. O aspecto `HybridMigrationMain` é um exemplo de como é utilizado:

```
trait HybridMigrationMain extends IMain{

       //before - main
       abstract override def main(args:Seq[String]) = {
         println("Main before")
         try { MPI.Init(args.toArray) ;}
         catch{case e:MPIException =>e.printStackTrace()}
         println("INIT MPI");
         super.main(args)
       }
}
```

Nesta implementação todos os aspectos são implementados por *traits*, embora a estrutura seja diferente. Na secção implementação será explicado com pormenor.

7.2.7 Avaliação

Uma das vantagens que traz o uso de *mixins* é que permite a reutilização dos *traits*. Se existirem outras classes com métodos iguais (mesmo nome, número e tipo de argumentos e valor retornado), podemos misturar os mesmos *traits* e ter a certeza que o comportamento original será alterado, trazendo vantagens na reutilização e na composição modular, um dos objectivos deste trabalho.

Outra vantagem da utilização de *mixins* é que é fiel ao princípio do AspectJ – e como tal, do POA – de alterar o comportamento do código fonte sem ter alterar o código fonte – o princípio de *obliviousness*. Basicamente desde que os métodos, aos quais queremos modificar o comportamento, estejam presentes na interface de serviços, podemos misturar *traits* para alterar ou adicionar comportamento.

Um dos problemas deste método é que não permite capturar detalhes específicos no código base – o facto de não ter suporte para os *pointcut designators*, torna a tarefa difícil, e embora seja possível recorrer a algumas técnicas para obter esses detalhes, não será um sistema simples como o AspectJ apresenta e não tão completo.

Um outro problema desta técnica é que, apesar de existir interoperabilidade entre Scala e Java, este último não possui mecanismo de *traits* e, como tal, não os reconhece, mas sim limita-os a interfaces. Embora, como vimos na secção 5.2.2, o compilador do Scala "contorne" a situação, e permita, de certa forma, que um *trait* seja reconhecido por uma classe Java e seja possível aceder aos métodos definidos naquele através da classe auxiliar, é uma solução improvisada, que não oferece a mesma ligação natural entre classes Scala e *trait*, e se houver alterações aos métodos neste é necessário também alterações na classe estendida o que não é muito prático para o que se pretende.

Na emulação do aspecto *HybridMigration*, não foi possível implementar dois dos *advices*. Isto aconteceu uma vez que o ponto do código que se pretendia capturar era a execução de métodos em objectos. Como foi explicado, existe uma diferença entre Java e Scala, na definição de membros e métodos estáticos. Enquanto que na primeira é permitido defini-los em classes regulares, na última estes têm de ser definidos em módulos especiais chamados object (Odersky et al, 2008). Acontece que os pontos de execução que se pretendem capturar são de métodos que estão em objectos que são utilizados por outros aspectos, ou seja, pretende-se capturar pontos não do código fonte mas já de aspectos, algo que o AspectJ permite fazer. Usando a técnica de *mixins* não é possível capturá-los, da forma como tem sido feita, pela razão de que um objecto nunca é instanciado, ele é acedido directamente. Para a técnica funcionar, é necessário misturar o aspecto com a classe *Core* no momento da instanciação, logo nunca irá funcionar com objectos. O mesmo problema ocorre se se quiser capturar métodos estáticos em Java: não seria possível capturá-los dado que para invoca-los não é necessário instanciar a classe, logo nunca será possível fazer um *mixin* com um aspecto.

7.3 Comparação Scala e AspectJ

Em AspectJ, os *crosscutting concerns* de paralelização do JECoLi, foram implementados usando aspectos. Dentro do Scala, em cada técnica, estes *concerns* também foram implementados por módulos próprios da linguagem, embora de forma e com uso diferente.

7.3.1 AspectJ e ScalaMixins

Usando Scala Mixins, foram usados *traits*. A natureza do uso de *mixins* levou a que um aspecto fosse dividido (ou particionado) em vários *traits* diferentes. Quando um *trait* é misturado com uma classe, existe a possibilidade de acrescentar comportamento à classe, através da introdução de novos métodos, ou a de alterar o comportamento, através do uso de *override*. Como é necessário que ambos partilhem uma interface comum, os métodos nos *traits* precisam de ser abstractos. O que acontece é que se tiver métodos abstractos e que alterem o comportamento de métodos que não existam na classe *core*, dá origem a erro de compilação. É natural que em AspectJ, um aspecto possa interceptar métodos de classes diferentes sem conflitos. Mas usando ScalaMixins, a regra é: por cada classe cujo comportamento se quer alterar e, se esta implementa uma interface diferente da que já existe num *trait*/aspecto, então é preciso criar um novo. Ou seja, é possível reutilizar um *trait*, mas apenas com classes que implementem a mesma interface (ou as mesmas interfaces).

Esta relação entre *traits* e classes, mostra-nos mais uma diferença, que se vê na forma como o aspecto/*trait* é utilizado. Enquanto que em AspectJ, este funciona como uma instância única e não necessita de ser criado e inicializado (aspecto não possui construtor), já em nos ScalaMixins o *trait* é misturado com uma classe no momento em que é criado uma instância desta, formando um só com a mesma. Ou seja, se criarmos 20 instâncias de uma classe, vamos misturar 20 vezes o *trait* com cada uma (se quisermos alterar o comportamento das mesmas, claro). Isto significa que é necessário ter o cuidado de fazer *mixin* sempre com uma nova instância da classe. Em relação ao estado do aspecto, em AspectJ é apenas um e pode ser alterado e usado conforme vai capturando pontos de junção e executando *advices*. Já em Scala, dado que misturamos um *trait* com uma classe, este vai ter estado próprio juntamente com a classe. Ou seja, esta junção é muito parecida com declarações intertipo (Laddad, 2012) – acrescenta-se membros e métodos a uma classe e passam a fazer parte dela.

Se tivermos duas instâncias da mesma classe, misturadas com o mesmo *trait*, estes podem apresentar estados diferentes. No caso do ParJeCoLi e da implementação feita em AspectJ torna-se prejudicial, porque existem *advices* cuja execução depende de dados obtidos por outros. Se estes *advices*, representados pelos métodos, estiverem em *traits* diferentes (e logo vão ser misturados com classes diferentes), não vai ser possível aceder ao estado e à informação pretendida. Para resolver este problema, criou-se um objecto comum que guarde a informação necessária a ser partilhada entre eles. O exemplo em Scala embaixo explica perfeitamente esta parte:

```
trait SharedMigrationConstructor extends IConstructor{

	//before -- poincut argumentos
	abstract override def createMultiGAPars(np:Int, itsAuton:Int,
numMigrants:Int, sync:Boolean):MultiGAPars={
		(…)
		SharedMigration.synchronized = sync//INFORMAÇÃO GUARDADA NO
OBJECT SHAREDMIGRATION
		if(SharedMigration.synchronized) //INFORMAÇÃO GUARDADA NO OBJECT
SHAREDMIGRATION

		{
			SharedMigration.barrier=new CyclicBarrier(np);
//INFORMAÇÃO GUARDADA NO OBJECT SHAREDMIGRATION
		}
		return super.createMultiGAPars(np, itsAuton, numMigrants, sync)
	}
   }
```

Este *trait*, implementa um dos *advices* do aspecto SharedMigration. O que acontece é que ele vai ser misturado com uma classe Scala (a classe Constructor, ver secção 7.2.4) e vai obter informação do contexto. Essa informação vai usada por um outro método, num *trait* diferente que implementa um *advice* diferente do SharedMigration:

```
trait SharedMigration [S <: IConfiguration] extends AbstractAlgorithm[S] with
AbstractMigration {

   //before -- pointcut migra
   abstract override def iteration(algorithmState:AlgorithmState,
solutionSet:ISolutionSet):ISolutionSet = {
	(…)
	if(SharedMigration.synchronized){ //INFORMAÇÃO OBTIDA DO OBJECT
   SHAREDMIGRATION

	(…)
		try {
		  putInBuffer(algorithm,algorithmState);
SharedMigration.barrier.await();//INFORMAÇÃO OBTIDA DO OBJECT SHAREDMIGRATION

		(…)
	}
      super.iteration(algorithmState, solutionSet)
}

object SharedMigration{
   var barrier:CyclicBarrier = null
   var synchronized=true
   }
```

No *trait* SharedMigration, vai ser usada a informação obtida pelo *trait* anterior. Se essa informação não for obtida, o que vai acontecer é que vai ocorrer erro de excepção, porque a

informação não está disponível. O objecto SharedMigration funciona como reservatório, onde a informação é partilhada pelos *traits* que na prática, em AspectJ, compõem o mesmo aspecto.

O modo de funcionamento do *trait*, relativamente ao de um aspecto na linguagem AspectJ, não significa que seja um problema ou uma desvantagem. A abordagem que esta técnica apresenta, permite obter alguma reutilização, no sentido em que os *traits* podem ser utilizados com qualquer classe que implemente a mesma interface (e tenha os métodos, que o *trait* modifica, definidos, caso contrário dá erro de compilação). Um outro ponto é que podemos seleccionar as instâncias com que queremos fazer *mixin*, ou seja, se quisermos alterar o comportamento de apenas algumas, é possível, basta apenas misturar as que queremos com o *trait*. Esta solução pode ser interessante em alguns casos. Em AspectJ, a forma mais fácil de ser conseguir isto seria utilizar o PCD *within* (ver secção 2.1), embora esta solução não resolva todas os casos, porque apenas limita ou exclui a captura da invocação de um método da instância dentro um package ou classe. Se tivermos as instâncias a correr na mesma classe ou package, torna-se mais problemático limitar a acção do aspecto sobre. Em alternativa poder-se-ia utilizar o PCD target e pelo estado do objecto distinguir se é um que interessa alterar o comportamento ou não. Para isso seria necessário uma marca, uma variável que funcionasse um pouco como "número de série". Seria talvez trabalhoso porque seria necessário definir condição dentro de um *advice* para excluir instâncias que não interessam e poderia trazer algum overhead em casos em que tenhamos múltiplas instâncias, uma vez que todas seriam interceptadas se fossem invocadas naquele dado ponto, quando o método ou construtor fosse chamado.

Em relação a *pointcuts* e *advices*, estes não existem nos *traits*. Relativamente aos *pointcut designators call* e *execution*, não existe distinção entre ambos dentro da técnica e estes são implementados naturalmente dentro do *trait*, quando declaramos um método para alterar o comportamento de um método da classe core. Nem todos os PCD´s foram emulados pela razão de que nem todos foram usados na implementação do ParJeCoLi. Mais a frente será falado sobre este ponto, dado que a implementação destes é comum em ambas as técnicas.

Em ScalaMixins, a ordem de precedência é a ordem de mistura com a classe *core* (ver secção 7.2.3). Em AspectJ é possível definir a ordem de precedência em um aspecto.

7.3.2 DSLScala e ScalaMixins

Existem alguns pontos em comum entre ambas as técnicas e são descritas aqui e comparadas com o AspectJ. Em relação aos *pointcut designators*, no ParJeCoLi, existem dois que foram implementados da mesma forma em ambas as técnicas: o PCD *within* e *this*.

Da mesma forma, para implementar intertipos, ambas as técnicas utilizam *traits* para emular o funcionamento (ver secção 7.1.97.1.9).

8. Validação

A validação das implementações é feita em três dimensões, todas elas comparando com a implementação do ParJECoLi em AspectJ:

- Completude - foi possível emular todos os aspectos presentes no ParJECoLi?
- Corretude – as soluções obtidas estão correctas (ou se são próximas das obtidas no ParJECoLi original)? A *framework* funciona como é suposto?
- Desempenho - o tempo de execução é igual ou diferente, e no caso deste último, muito ou pouco?

A primeira dimensão é fácil de validar, na medida em que foi possível verificar, durante a implementação, se houve alguma parte de algum aspecto que foi possível implementar ou não. No caso da DSLScala, todos os aspectos foram emulados sem problema, ou seja, para todos os mecanismos AspectJ utilizados, foi possível chegar a uma solução que os implementasse. Por exemplo, foi possível emular o funcionamento de todos os *pointcut designators* (para relembrar, ver 2.1) usados na *framework*. Já em relação à solução ScalaMixins, houve partes de alguns aspectos que não foi possível implementar, pelo que esta solução falhou em parte no objectivo. A limitação desta solução dá-se no caso em que se pretende interceptar chamadas a métodos estáticos, o que acontece em alguns dos aspectos.

Para avaliar a duas dimensões a seguir, foi necessário usar um caso de estudo. Tendo em conta que o ParJECoLi é uma *framework*, a única forma de verificar se o funcionamento está correcto e o tempo de execução dentro do aceitável, foi introduzir um problema e, a partir daí, comparar os resultados. O caso de estudo utilizado é um caso que procura optimizar o processo de fermentação de uma bactéria. Este caso foi usado em investigações anteriores (Pinho, Sobral e Rocha, 2012), para demonstrar a funcionalidade e o sucesso do uso do AspectJ em paralelizar a plataforma JECoLi, adaptando-a para correr em ambientes de computação paralela.

Para verificar se o ParJECoLi, implementado em Scala, funciona como pretendido, torna-se necessário comparar o seu comportamento com o da implementação em AspectJ, nomeadamente se corre os modelos de paralelismo pretendidos mas, acima de tudo, se produz resultados iguais ou semelhantes. Neste caso de estudo, não existe uma única resposta correcta, uma vez que usa algoritmos para procurar a melhor resposta ou solução, dado um

problema. Como tal, a única solução é comparar se os resultados obtidos se aproximam ou não da implementação em AspectJ. Este caso de estudo foi testado em dois modelos de paralelismo, mencionados no capítulo 3, secção 3.4.1:

- Avaliação dos vários indivíduos que compõem uma população é feita de forma paralela, ou seja, a população é dividida consoante o número de *threads*, definidas no ficheiro de configuração, que por sua vez vão avaliar o *fitness* de cada um dos indivíduos (qualidade da solução). A função de avaliação é definida por quem introduz na *framework* o caso de estudo. A ideia principal é: se existirem 4 *threads*, numa arquitectura *multicore*, em teoria, cada uma das *threads* terá um *core* para si, o que significa que o trabalho que era realizado por um passa a ser realizados por quatro;
- O segundo modelo usado é o modelo de ilhas, também referido no capítulo 3. Este modelo pode criar uma ou mais populações, isoladas, entre as quais pode haver migração de indivíduos ou não. Nessa migração, passam um número definido de indivíduos (escolhidos aleatoriamente). O número de populações, a existência ou não de migração ou não, em que momento ela se dá, se é sincronizada entre todas as ilhas e o número de indivíduos que migram, tudo isso é definido num ficheiro de configuração. Este modelo tem mais como objectivo não o tempo de execução, mas sim o alcançar melhores resultados, melhores soluções para o problema;

Nos gráficos a seguir, são demonstrados os resultados obtidos e é feita uma comparação com o ParJECoLi, implementado usando AspectJ. Os testes foram corridos numa máquina *quad-core* i7-3610QM, 2.30GHZ, 6 GB Ram, usando o sistema operativo Ubuntu 14.04, versão 64 bits.

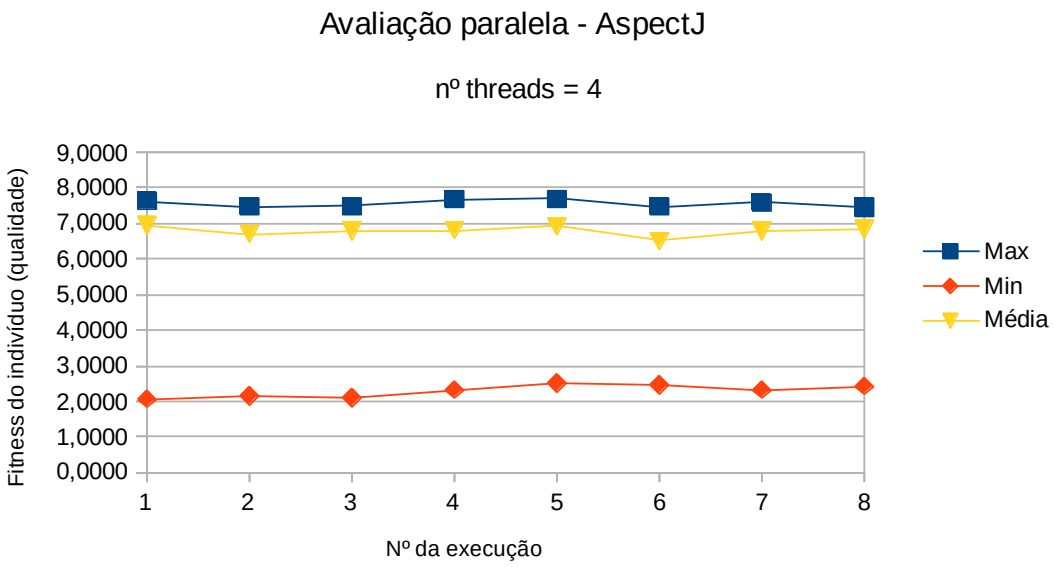

Figura 8. 1 - Resultados de execução, do caso de estudo de optimização do processo de fermentação, usando a versão ParJECoLi implementada com AspectJ. Modelo de paralelismo usado foi a avaliação

paralela de cada indivíduo, com quatro *threads*. O ponto *max* refere a melhor solução, o *min* a pior solução e a *média* refere a média do *fitness* da população

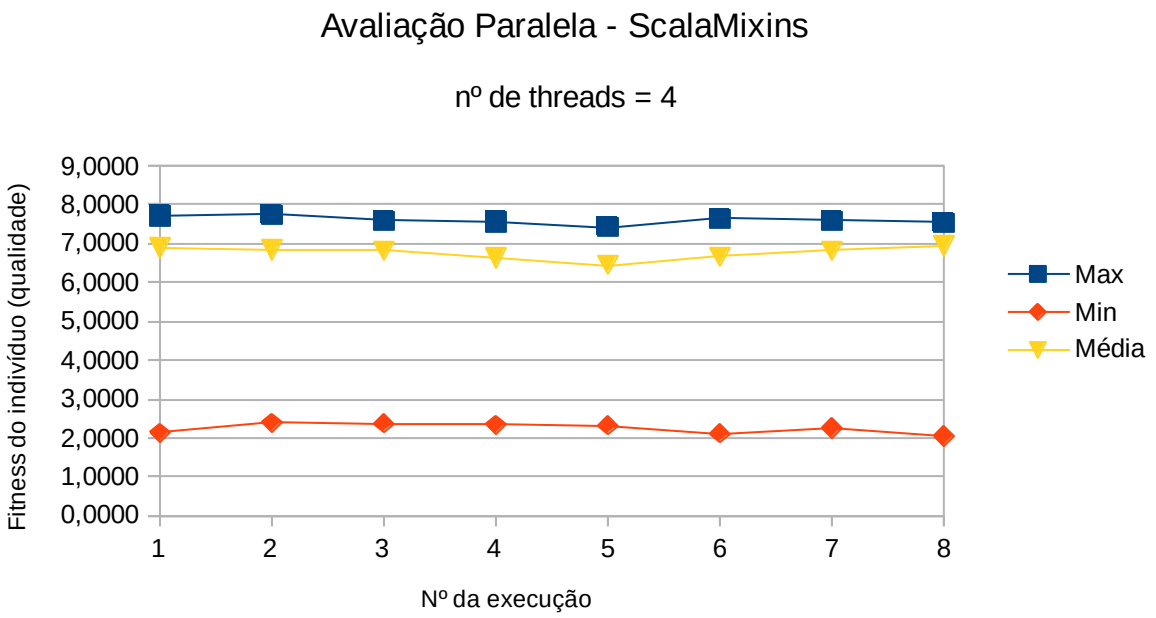

Figura 8. 2 - Resultados de execução, para o mesmo caso de estudo, nas condições mencionadas na figura 8.1, mas usando ParJECoLi implementado com ScalaMixins.

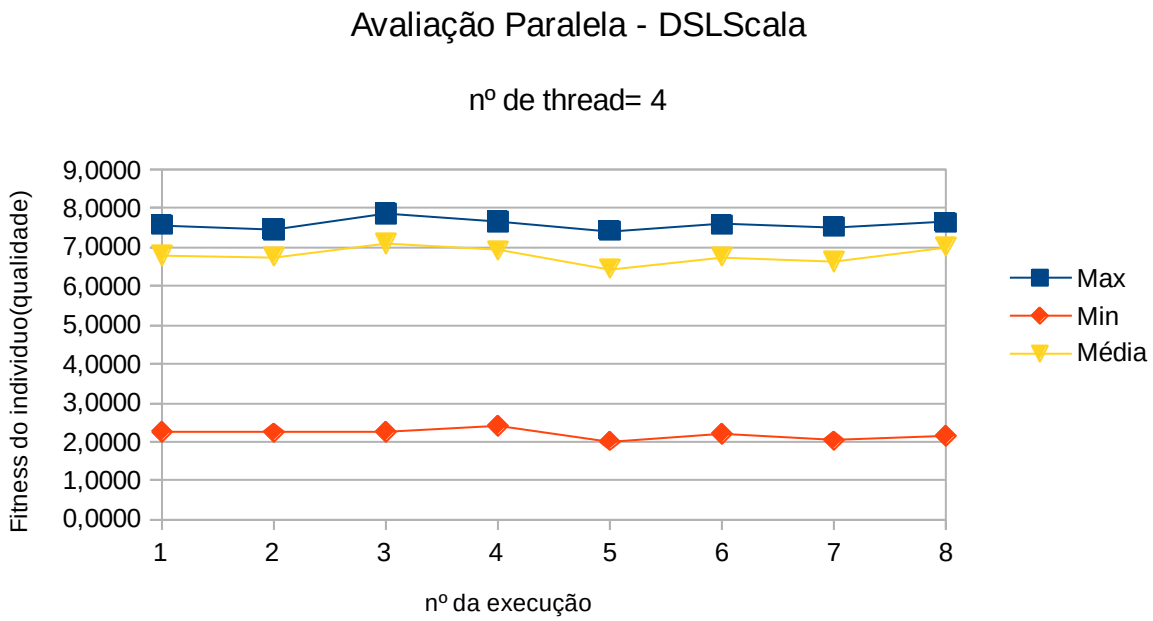

Figura 8. 3 - Resultados de execução, nas condições mencionadas nas Figura 8.1, mas usando a versão ParJECoLi implementada com DSLScala.

A Figura 8. 1, Figura 8. 2 e Figura 8. 3 demonstram os resultados de oito execuções do caso de estudo de optimização do processo de fermentação, usando o modelo de avaliação paralela da qualidade dos indivíduos da população. As figuras também indicam que foram usados, em todas as execuções, quatro *threads*, e indicam qual da versões do ParJECoLi foi usado. Os resultados podem ser interpretados da seguinte forma: os pontos *min*, indicado no gráfico, representam o indivíduo, em cada execução, que segundo a função da avaliação, é a menor solução, enquanto que os pontos *max* indicam o contrário. A média indica a qualidade de soluções, dentro da população, depois do algoritmo chegar ao fim.

Podemos verificar, comparando o gráfico do AspectJ com os gráficos das versões ScalaMixins e DSLScala, que as soluções obtidas são muito semelhantes. Tanto o valor máximo, como o mínimo e a média, encontram-se muito próximo em cada uma das versões, pelo que, em relação a este caso de estudo, se pode concluir que o funcionamento é o pretendido.

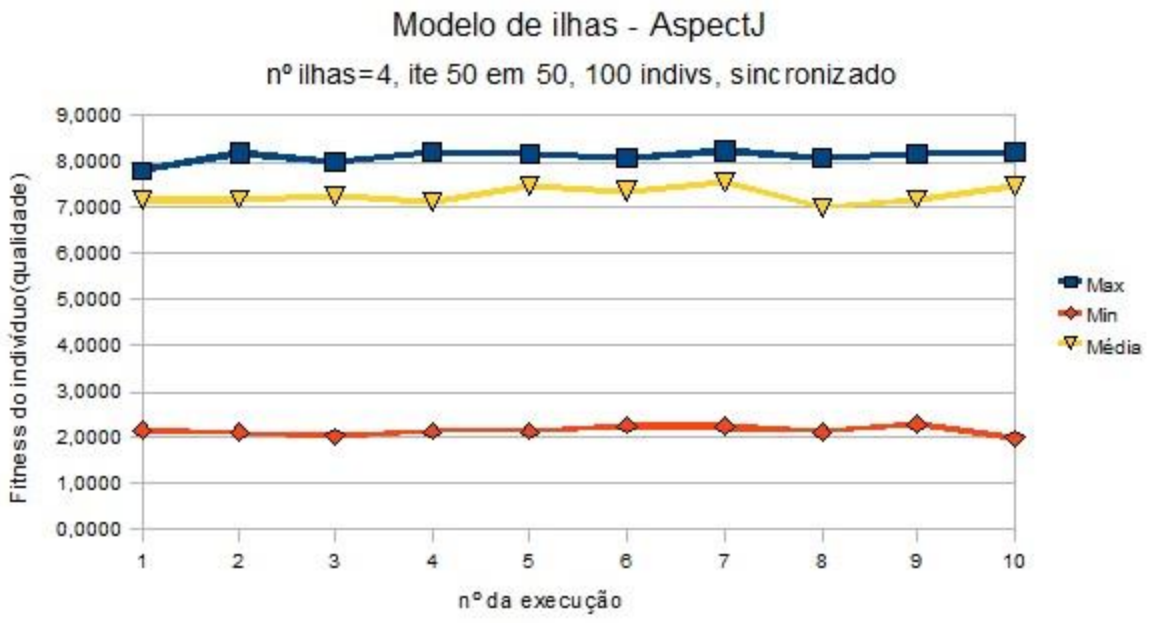

Figura 8. 4 - Resultados da execução, do caso de estudo, com ParJECoLi em AspectJ.

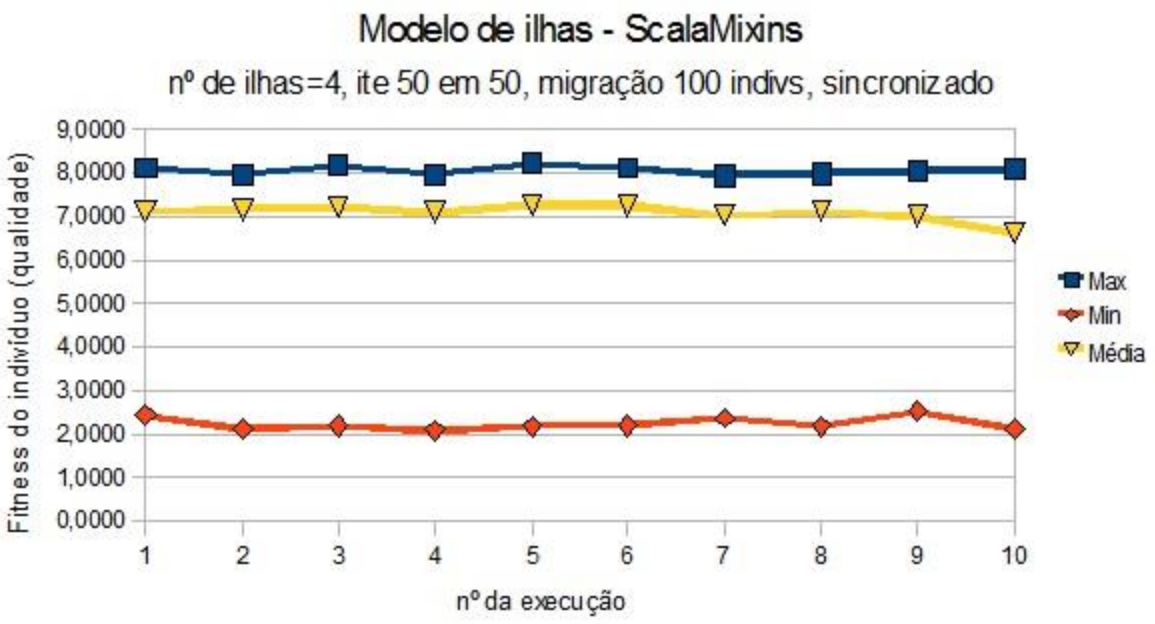

Figura 8. 5 - Resultados da execução, do caso de estudo, usando a versão em ScalaMixins.

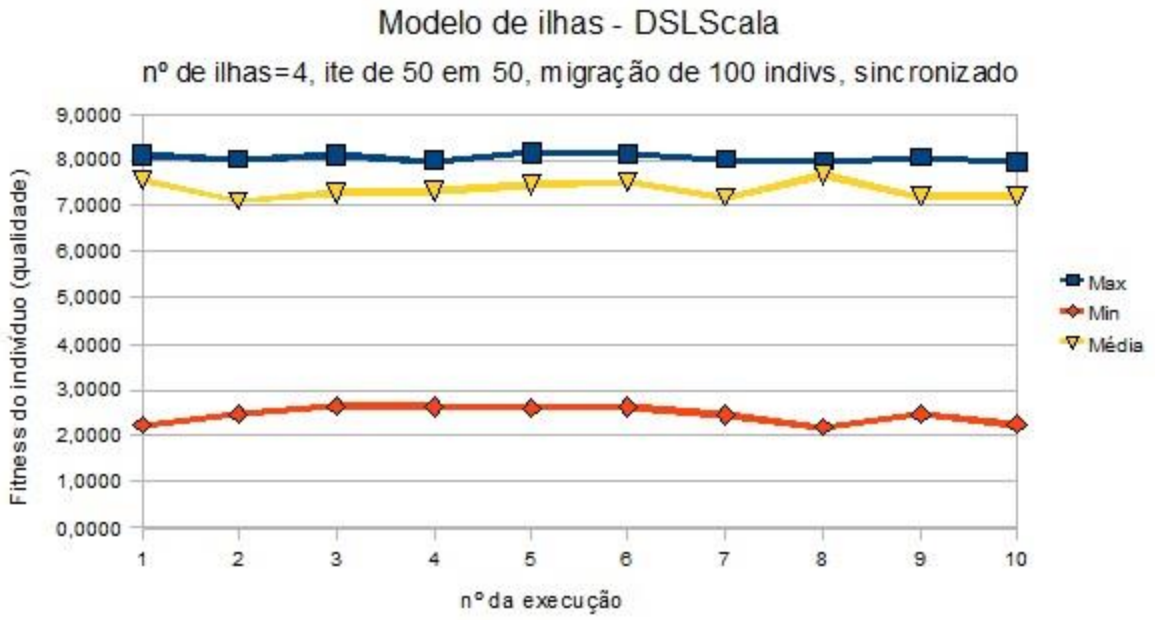

Figura 8. 6 - Resultados da execução, do caso de estudo, usando a versão em DSLScala.

A Figura 8. 4, Figura 8. 5 e Figura 8. 6, mostram a execução do caso de estudo, dentro de um outro modelo de paralelismo, o modelo de ilhas. Conforme indicado nos gráficos, o número de ilhas criadas em cada execução foram 4, com migração a cada 50 iterações, de 100 indivíduos. A migração entre as ilhas decorreu de forma sincronizada. Igualmente pelos resultados, tanto do maior valor, como do menor ou dá media do *fitness* dos indivíduos,

podemos ver que as soluções aproximam-se, o que significa que também neste modelo, a *framework* implementada nas soluções Scala, funciona como pretendido.

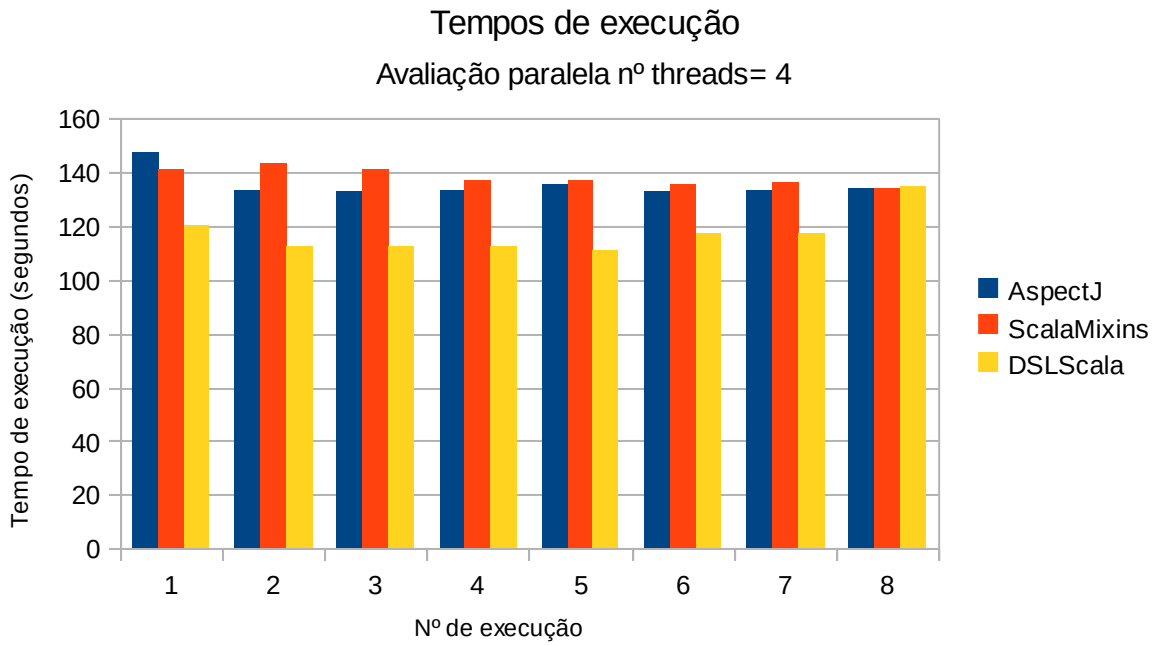

Figura 8. 7 - Comparação dos tempos de execução, para o modelo de avaliação paralela.

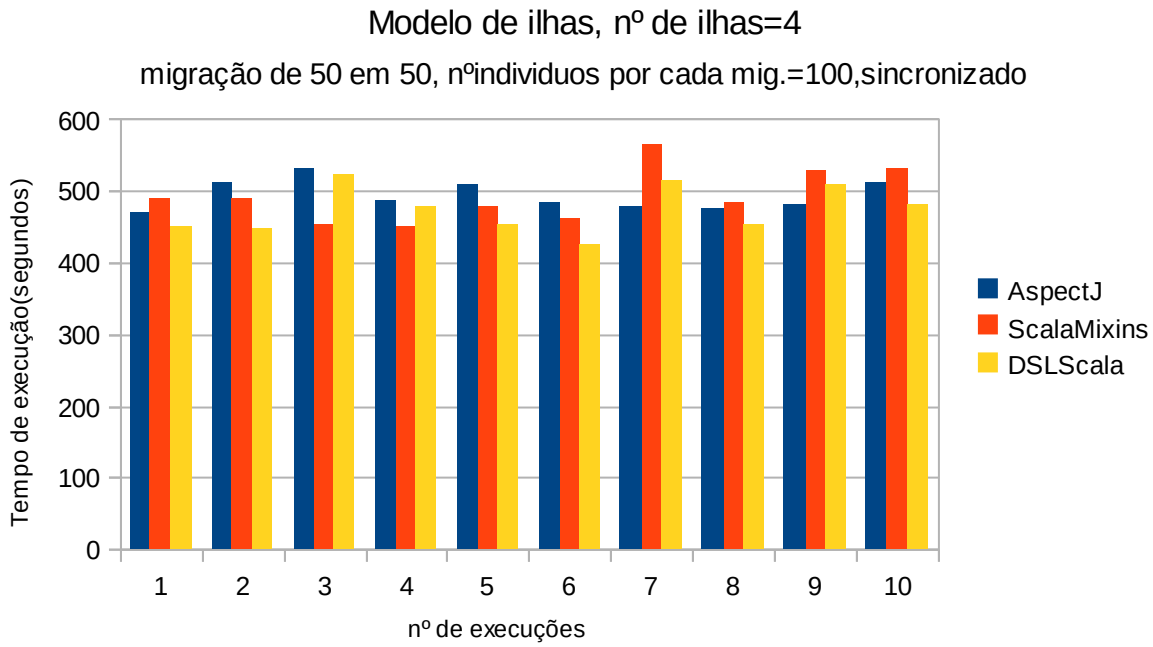

Figura 8. 8 - Comparação dos tempos de execução, para o modelo de ilhas.

Em relação ao tempo de execução, podemos ver, pela Figura 8. 7 e Figura 8. 8, que a diferença dos tempos de execução é mínima, o que sugere que a execução do caso de estudo numa *framework* implementada com Scala não sofre de nenhum problema ou complicação de execução. Nalguns casos, até se verificou que a execução do ParJECoLi implementada com Scala teve melhores resultados, embora não esteja excluído de ter havido alguma condição adversa não detectada, aquando da execução dos outros testes.

9. Conclusões e Trabalho Futuro

Perante os objectivos traçados, este trabalho permitiu chegar a algumas conclusões. Foi possível aferir a capacidade do Scala e verificar que apresenta construções e mecanismos que lhe permitem emular o funcionamento de AspectJ. A nova unidade de modularização sugerida pela POA, o aspecto, foi emulado usando, num caso, uma unidade própria do Scala, o *trait* e, num outro caso, classes Scala e *traits*, o que serviu para enriquecer ainda mais o estudo e ter uma noção mais exacta da sua capacidade. Numa das soluções, o funcionamento é mais similar ao de um aspecto em AspectJ, no sentido em que existe uma única instância que vai adicionar ou alterar o comportamento do código base. Nessa solução, o facto de um *trait* ter estado próprio e métodos definidos, permitiu uma composição modular diferente. Na outra abordagem, é usada uma outra capacidade do *trait*, que é o facto de se poder mistura-lo com classes, para alterar o comportamento. Neste caso, não funciona exactamente como um aspecto funciona em AspectJ, mas o comportamento, o objectivo, mantém-se: é adicionado comportamento ao código base ou é alterado o mesmo. Esta nova abordagem, trazida pelo ScalaMixins, permite uma composição modular diferente entre aspecto/*trait* e classes.

Uma das limitações do AspectJ, referido em (Ongkingco et al, 2006), prende-se com o facto de um aspecto estar fortemente acoplado a uma dada classe, o que faz com que viole um dos princípios da modularidade: o de esconder a implementação por detrás de uma interface bem definida. Nas abordagens utilizadas, só uma delas não viola este princípio. Em ScalaMixins tanto a classe como o *trait* (que funciona como aspecto) partilham uma mesma interface. O que significa que qualquer alteração no nome do método ou em argumentos, tem como consequência, uma alteração na interface. Isto vai forçar a que essa alteração também seja feita no *trait*. Para além disso, podemos afirmar que todos os aspectos têm a sua implementação escondida numa interface. Em DSLScala, dado o caso de estudo utilizado, não houve necessidade de usar interfaces para representar as classes que emulam o funcionamento de um aspecto. Ainda assim, uma vez que são usadas classes, estas permitem a implementação de interfaces. No entanto, verifica-se que o problema, que existe em AspectJ, também existe nesta abordagem: um aspecto está fortemente acoplado a uma classe. A necessidade de se ter de especificar, num *pointcut*, a classe do método que se pretende interceptar e não uma interface, limita a sua reutilização. É possível, no entanto, dar a volta a esta situação, se conseguirmos obter as interfaces associadas a uma classe e, mudando algumas partes do código.

Um outro pormenor existente em AspectJ, e algo perigoso, reside em se poder declarar um *advice* que actua sobre o código fonte em qualquer aspecto. De certa forma, isto significa que tudo é permitido, não existem restrições. Em ScalaMixins, existe alguma restrição no sentido em que é necessário misturar o *trait*/aspecto com a instância de uma classe, para conseguir o efeito POA. Já em DSLScala, o funcionamento é muito parecido ao do AspectJ, pelo que esta liberdade, de certa forma, mantém-se.

A utilização do ParJECoLi neste trabalho funcionou em dois sentidos, num primeiro para comparar se o Scala conseguia emular os aspectos implementados em AspectJ, e num segundo, para testar se a versão do ParJECoLi, implementada em Scala, tinha o mesmo desempenho que o AspectJ, correndo um caso de estudo tanto na versão AspectJ como na versão em Scala (neste caso, para as duas abordagens utilizadas). A utilização do ParJECoLi como domínio do problema, permitiu também verificar que o Scala se apresenta como uma alternativa no suporte a computação paralela. O facto de ter emulado o funcionamento de aspectos que implementam *crosscutting concerns* de paralelização, e, posteriormente, verificar que as soluções e desempenho obtidos são próximos e, nalguns casos, até melhor do que os obtidos em AspectJ, leva a que possamos considerar que o Scala permite esse suporte. Eventualmente, em trabalho futuro, será possível ter alguma noção até que ponto conseguirá oferecer soluções para todas as exigências da Computação Paralela.

Pela comparação dos resultados obtidos, no capítulo 8, podemos concluir que o uso de Scala para emular AspectJ não é uma solução que limite de algum modo o desempenho da *framework*, pelo que se apresenta como uma opção credível, dentro destes termos, para dar suporte a computação paralela.

Nas soluções apresentadas, foram verificadas algumas limitações, nomeadamente na abordagem ScalaMixins, onde não é possível interceptar a chamada a métodos estáticos, isto é, métodos definidos em objectos, uma vez que estes não possuem instâncias e, como tal, não é possível realizar mistura com *traits*. Uma possível solução para esta limitação será o de utilizar o modo de funcionamento da abordagem DSLScala, para interceptar as chamadas a métodos estáticos, ou seja, delegar o código do objecto para uma terceira parte. Se existir algum aspecto interessado em executar código antes ou depois, teria de estar associado de alguma forma – utilizar o mecanismo de *pointcuts* proposto é uma hipótese. Pensou-se nesta solução para resolver este problema, no entanto, uma vez que se pretende estudar as técnicas de forma separada, com as suas vantagens e desvantagens ou limitações, não se chegou a implementar esta solução.

Seria interessante criar uma espécie de modelo híbrido entre as duas soluções e de realizar um estudo para verificar em que casos é mais viável utilizar uma ou outra opção. Tendo em conta que a composição modular é diferente em ambos os casos, poderá ser

interessante ver como será possível viabilizar o funcionamento em conjunto de ambas as soluções, no sentido de conseguirem implementar *crosscutting concerns*, e se o funcionamento em conjunto apresenta um leque mais alargado de opções que permitam implementar todos os casos ou maior flexibilidade ao programador para implementar um caso mais do que uma forma, consoante lhe for mais favorável.

Na solução DSLScala foi referido o problema de não ser possível estabelecer uma ordem de precedências. Existem formas de contornar esta situação mas dado que nenhum caso no ParJeCoLi exigia um mecanismo assim, não houve a preocupação de implementá-lo. No entanto, seria interessante num trabalho futuro permitir que tal fosse possível.

Uma coisa que também se pode concluir é que, apesar de a interoperabilidade entre Java e Scala ser possível, ela apresenta algumas limitações:

- no uso de *traits* – o seu uso encontra-se limitado, uma vez que o Java reconhece-os como *interfaces*, o que retira uma parte das suas capacidades. Um *trait* tem de ser implementado (como uma interface comum) e isto significa que qualquer método já definido, tem de voltar a ser implementado na classe.
- em efectuar *mixins* - quando se pretende criar uma instância de uma classe e se quer realizar uma mistura com um *trait*, tal não pode ser feito numa classe Java, apenas em classes Scala. Pode-se contornar esta limitação, como foi feito com o ParJECoLi, delegando essa função para uma classe Scala, mas tal torna-se algo invasivo.
- no uso de funções de ordem elevada – em Scala torna-se menos invasivo o uso deste mecanismo. O corpo de um método pode ser passado como argumento de um método de uma forma muito simples e eficaz. Em Java, é necessário acrescentar bastantes linhas para conseguir o mesmo resultado, tornando o código algo confuso.
- nos tipos parametrizados - o facto de não se pode usar *raw types* em Scala. Todos os tipos parametrizados têm de estar definidos quando se cria um objecto ou se declara um método.

Estas limitações, levam a que a composição natural entre classes Java e *traits* não seja tão natural como acontece em Scala. Tendo em conta o trabalho realizado, esta dissertação é um contributo para aferir as limitações de operabilidade entre ambas as linguagens. Face aos problemas experimentados, uma sugestão para trabalho futuro seria o de usar uma aplicação ou *framework* implementada em Scala e tentar adaptá-la para correr em ambientes de computação paralela usando as soluções apresentadas neste trabalho. Igualmente, seria interessante usar os mecanismos que o Scala disponibiliza, como actores, tipos Future, para implementar modelos de paralelismo. Uma hipótese também a ter em conta é a de usar a biblioteca Akka, referida na secção 6.1, e verificar até que ponto é invasiva, usando os

mecanismos e técnicas que disponibiliza, se produz ou não os efeitos de emaranhado e dispersão de código.

O trabalho produzido nesta dissertação pode servir como base para, no futuro, desenvolver uma biblioteca reutilizável. É necessário resolver ainda algumas das limitações das soluções e, verificar com mais casos de estudo, se o Scala se apresenta como uma solução mais viável para paralelizar aplicações em Java, em Scala ou em ambas. Igualmente, depois de resolvidas as limitações, verificar se é possível implementar os mecanismos de computação paralela de uma forma abstracta, em seja possível aplica-los ao código fonte que se pretende paralelizar, conforme pretendido em (Sobral, Cunha e Monteiro, 2006a) e (Sobral, Cunha e Monteiro, 2006b).

10.Bibliografia

(Akka, 2014), "*Akka*", http://akka.io/, ultimo acesso Novembro 2014

(Alba e Tomassini, 2002), E. Alba, M. Tomassini, "*Parallelism and Evolutionary Algorithms*", IEEE Transactions on Evolutionary Computation, Vol 6, nº5, October 2002.

(Andrási, 2013), S. Andrási, "*Understanding Scala's partially applied functions and partial functions*",http://sandrasi-sw.blogspot.pt/2012/03/understanding-scalas-partially-applied.html, ultimo acesso em Dezembro de 2013.

(AspectJ, 2014) "*The AspectJ Programming Guide*", https://www.eclipse.org/aspectj/doc/next/progguide/, ultimo acesso: Setembro de 2014

(AspectJb, 2014) "*AspectJ Frequently Asked Questions*" http://www.eclipse.org/aspectj/doc/released/faq.php, ultimo acesso: Setembro de 2014

(Belding, 1995), T.C. Belding, "*The Distributed Genetic Algorithm Revised*", in Proc. 6th Int. Conf. Genetic Algorithms, L.J. Eshelman, Ed. 1995, pp. 114-121.

(Belvins, 2010), M. Belvins, "*Dynamic Method Interceptors in Scala*", retirado de http://cleverlytitled.blogspot.pt/2010/01/dynamic-method-interceptors-in-scala.html.

(Barney, 2014), B. Barney, "*Introduction to Parallel Computing*", Lawrence Livermore National Laboratory, ultimo acesso: Dezembro 2014

(Bonér, 2008), J. Bonér, "*AOP-style Mixin Composition Stacks in Scala*" em http://jonasboner.com/2008/02/06/aop-style-mixin-composition-stacks-in-scala/, ultimo acesso: Junho de 2013

(da Cruz, 2003), F. da Cruz, "*Columbia University Computing History: The IBM 704*". Universidade de Columbia, 2008-01-08.

(Evangelista et al, 2009) "*A Software Platform for Evolucionary Computation with Pluggable Paralelism and Quality Assurance*", 2009

(Evangelista, Maia e Rocha, 2009), P. Evangelista, P. Maia, M. Rocha, "*Implementing metaheuristic optimization algorithms with JECoLi*", Intelligent Systems Design and Applications, 2009, Pisa.

(Flynn, 1972), M.J. Flynn, "*Some Computer Organizations and Their Effectiveness*", IEEE Transactions on Computers, Volume 21 issue 9, Setembro 1972.

(Filman e Friedman, 2000), R.E. Filman, D. P. Friedman, "*Aspect-Oriented Programming is Quantification and Obliviousness*", Workshop on Advanced Separation of Concerns at OOPSLA 2000, Minneapolis, October 2000

(Gamma et al, 1994), E. Gamma, R. Helm, R. Johnson, J. Vlissides, "*Design Patterns: Elements of Reusable Object-Oriented Software*", Addison-Wesley, Novembro de 1994.

(Gil, 1958), S. Gill, " *Parallel Programming*", The Computer Journal Vol. 1 #1, pp2-10, British Computer Society, April 1958.

(Gonçalves e Sobral, 2009), R.Gonçalves, J. Sobral, "*Pluggable Parallelization*", High Performance Distributed Computing, Munique, ACM Press, Junho 2009.

(Gradecki et Lesiecki, 2003), J. Gradecki, N. Lesiecki, "*Mastering AspectJ: Aspect-Oriented Programming in Java*", Wiley Publishing Inc, 2003.

(Groop et al, 1996), W. Groop, E. Lusk, A. Skjellum, "*A High-Performance, Portable Implementation of the MPI Message Passing Interface*", Parallel Computing, 1996

(Harbulot, Gurd, 2004), B. Harbulot, J. Gurd, "Using AspectJ to Separate Concerns in Parallel Scientific Java Code", Lancaster, AOSD 04, Março de 2004.

(Hargreaves et Merkle, 2013), Felix P. Hargreaves, D. Merkle, "FooPar: A Functional Object Oriented Parallel Framework in Scala", em PPAM, número 8385, em LNCS, páginas 118-129, 2014.

(Gottlieb et Almasi, 1989) A. Gottlieb, G.S. Almasi, "*Highly Parallel Computing*",

(JDK8, 2014), http://openjdk.java.net/projects/jdk8/, ultimo acesso: Janeiro 2014

(Kunz, 1991) Paul F. Kunz, "*Object Oriented Programming*", Stanford University, Agosto 1991.

(Laddad, 2012), R. Laddad, "*AspectJ: In Action, Enterprise AOP With Spring Applications*", 2nd Edition, Manning, 2010.

(Lea, 1999), D. Lea, "Concurrent Programming in Java", second edition, Addison-Wesley, 1999.

(Lin e Snyder, 2009), C. Lin, L. Snyder, "*Principles of Parallel Programming*", Pearson Internacional Edition. 2009.

(Macbeath, 2014), J. Macbeath http://jim-mcbeath.blogspot.pt/2009/08/scala-class-linearization.html#rules, ultimo acesso em Dezembro de 2013

(Monteiro e Fernandes, 2011), M. Monteiro, J. Fernandes, "*Aspect-oriented Refactoring of Java Programs*", 2011

(Meyer, 1997), B. Meyer, "*Object-Oriented Software Construction*", second edition, Prentice Hall, 1997.

(Odersky et al, 2008) M. Odersky, L. Spoon, B. Venners, "*Programming in Scala*", 2nd Edition, Artima Press, 2008

(Ongkingco et al, 2006) N. Ongkingco, P. Avgustinov, J. Tibble, L. Hendren, O. de Moor, G. Sittampalam, "*Adding Open Modules to AspectJ*", Bonn, Alemanha, AOSD 06, 20-24 de Março de 2006.

(Scala, 2013) http://www.scala-lang.org/, último acesso: Junho de 2013

(ScalaApi, 2013), http://www.scala-lang.org/api/current/#package último acesso Novembro 2013

(Spiewak, 2013), D. Spiewak, http://www.codecommit.com/blog/java/interop-between-java-and-scala, ultimo acesso em Novembro de 2013

(Spiewak e Zhao, 2009), D. Spiewak, Tian Zhao, "*Method Proxy-Based AOP in Scala*", Journal of Object Technology, Volume 8, Nº7, Novembro-Dezembro 2009.

(Sobral, 2006) , J. L. Sobral, "*Incrementally Developing Parallel Aplications with AspectJ*", IEEE IPDPS'06, Rhodes, Grécia, Abril de 2006.

(Sobral, Cunha e Monteiro, 2006a), J. L. Sobral, C. Cunha, M. P. Monteiro, "*Aspect Oriented Pluggable Support for Parallel Computing*", VecPar 2006, Rio de Janeiro, Junho de 2006

(Sobral, Cunha e Monteiro, 2006b), J. L. Sobral, C. Cunha, M. P. Monteiro, "*Reusable Aspect-Oriented Implementations of Concurrency Patterns and Mechanisms*", Bonn, Alemanha, AOSD 06, 20-24 de Março de 2006.

(Sobral, Monteiro e Cunha, 2006), J.L. Sobral, M.P. Monteiro, C. Cunha, "Comparison of Two Frameworks for Parallel Computing in Java and AspectJ", relatório técnico, Universidade do Minho, Portugal, Setembro de 2006.

(SETI, 2014), "*SETI Home*", setiathome.ssl.berkley.edu, ultimo acesso: Dezembro de 2014

(Pinho, Rocha e Sobral, 2010), J. Pinho, M. Rocha, J. Sobral, "*Pluggable Parallelization of Evolutionary Algorithms Applied to the Optimization of Biological Processes*", 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing, Pisa, Italy, February 2010

(Pinho, Sobral e Rocha, 2012), J. Pinho, J. Sobral, M. Rocha, "*Parallel evolutionary computation in bioinformatics applications*", Computer methods and programs in biomedicine, 2012.

(PVM, 2014), Parallel Virtual Machine, http://www.csm.ornl.gov/pvm/, ultimo acesso: Dezembro de 2014.

(Venners, 2009), B. Venners, "*Scala's Stackable Trait Pattern*", in Scalazine, Artima Developer articles "Best practices in Enterprise software development", 30 de Setembro 2009, http://www.artima.com/scalazine/articles/stackable_trait_pattern.html, último acesso em: Janeiro de 2014.

(Tanese, 1989), R. Tanese, "*Distributed Genetic Algorithms*", ICGA-3, Ed. 1989, pp. 434-439.

(Wyatt, 2012), D. Wyatt, "*Akka Concurrency*", PrePrint Edition, Artima Press.

Anexo A: Levantamento de mecanismos AspectJ do ParJECoLi

Nome do pointcut	Definido em que aspecto	Call	Target	Outros PCD e valores
argumentos	AbstractAlgorithm	MultiGAPars.**new**(**int**,**int**,**int**,**boolean**)	-	**args**(pops,iterationStep, migrants,syncro)
Migra	AbstractAlgorithm	ISolutionSet AbstractAlgorithm.iteration(AlgorithmState,ISolutionSet)	algorithm	**args**(state,solutions)
CathRun	BuildIslandModel	IAlgorithmResult IAlgorithm.run()	algorithm	!**within**(Island)

Tabela A. 1 - Levantamento de *pointcuts* com nome definidos no ParJECoLi

Tipo Advice	argumento(s)	Aspecto	Call	Target	Outros PCD e valores
Before	IAlgorithm algorithm	DistributedMigration	* IAlgorithm.run()	algorithm	within(Island)
	int pops,**int** iterationStep, **int** migrants, **boolean** syncro	DistributedMigration	Pointcut argumentos		
	IAlgorithm algorithm,AlgorithmState state,ISolutionSet solutions	DistributedMigration	Pointcut Migra		
	String[] args	HybridMigration	-	-	**args**(args) **execution**(* *.main(String[]))
	int pops,**int** iterationStep,**int** migrants,**boolean** syncro	SharedMigration	Pointcut argumentos		
	IAlgorithm algorithm,AlgorithmState state,ISolutionSet solutions	SharedMigration	Pointcut Migra		
	int pops,**int** iterationStep,**int** migrants,**boolean** syncro	SequentialMigration	Pointcut argumentos		
	IAlgorithm algorithm,AlgorithmState state,ISolutionSet	SequentialMigration	Pointcut Migra		

	solutions				
	IAlgorithm algorithm	Evaluation	IAlgorithmResult IAlgorithm.run()	algorithm	-
	IAlgorithm algorithm	TestEvaluate	IAlgorithmResult IAlgorithm.run()	algorithm	-
After	Island island,MigrationBuffer mb	AbstractMigration	**void** Island.addInBuffer(MigrationBuffer)	island	**args**(mb)
	Island island,MigrationBuffer mb	AbstractMigration	**void** Island.addOutBuffer(MigrationBuffer)	island	**args**(mb)
	GATopology gat	DistributedMigration	-	gat	**execution**(GATopology.**new**(*,*))
	MultiGA m	DistributedMigration	-	m	**execution**(MultiGA.**new**(*,*,*))
	-	DistributedMigration	-	-	**execution**(* IAlgorithm.run())
	MultiGA multiga	HybridMigration	-	multiga	**execution**(MultiGA.**new**(..))
	int pops,**int** iterationStep,**int** migs,**boolean** syncro	HybridMigration	Pointcut argumentos		
	ISolutionSet solutionset	HybridMigration	* SharedMigration.getFromBuffer(*,*,ISolutionSet)	-	**args**(*,*,solutionset)
	-	HybridMigration	-	-	**execution**(* IAlgorithm.run())
	-	TestEvaluate	-	-	**execution**(* IAlgorithm.run())
After returning	List<ISolution> list	HybridMigration	-	-	**execution**(* AbstractMigration.selectIndividuals(..))
	IAlgorithmResult resutl	HybridMigration	IAlgorithmResult IAlgorithm.run()	-	!**within**(Island)
Around	IEvaluationFunction eval,ISolutionSet solutionSet	Evaluation	**void** IEvaluationFunction.evaluate(ISolutionSet)	eval	**args**(solutionSet) !**within**(ThreadEvaluation)
	IAlgorithm algorithm	BuildIslandModel	Pointcut cathRun		
	MultiGA m	ParalelJeco	**void** run()	m	-
	String[] args	ParallelJecoMPI	-	-	**execution**(**public void** *.main(String[]))

					args(args)
	MultiGA m	ParallelJecoMPI	**void** run()	m	-
	ISolutionSet solutionSet	TestEvaluate	**void** IEvaluationFunction.evaluate(ISolutionSet)	-	**args**(solutionSet) !**within**(ThreadEvaluation)

Tabela A. 2 - Levantamento de *advices* e *pointcut designators* nos aspectos do ParJECoLi