Universidade Federal de Minas Gerais
Pós-Graduação em Ciência da Computação

União de Pontos de Junção em Linguagens Orientadas por Aspectos

Proposta de Dissertação de Mestrado

Proponente: Eduardo Santos Cordeiro

Orientador: Roberto da Silva Bigonha

Belo Horizonte, 13 de janeiro de 2006

Universidade Federal de Minas Gerais
Pós-Graduação em Ciência da Computação

União de Pontos de Junção em Linguagens Orientadas por Aspectos

Proposta de Dissertação de Mestrado

Proponente: Eduardo Santos Cordeiro

Orientador: Roberto da Silva Bigonha

Belo Horizonte, 13 de janeiro de 2006

Sumário

1 Definição do Problema

1.1 Contextualização

Programação orientada por aspectos (AOP, do inglês *Aspect-Oriented Programming*) é um modelo de programação que surgiu em 1997 com o objetivo de modularizar requisitos transversais [KLM+97]. Diz-se que dois requisitos se cruzam se a implementação ou o uso deles se tornam entrelaçados no código de um programa.

Embora normalmente os requisitos principais de grandes sistemas possam ser implementados de forma modular por meio de construtos existentes em linguagens orientadas por objetos, tais como métodos, classes e herança, a introdução nesses sistemas de requisitos secundários, embora vitais, muitas vezes atravessa as fronteiras de modularidade desse paradigma. Essa dificuldade gera maior complexidade e, conseqüentemente, maior custo de implementação e manutenção durante o ciclo de desenvolvimento desses sistemas.

A implementação mais notável dos conceitos de orientação por aspectos é a linguagem AspectJ [KHH+01], uma extensão de Java que oferece novas construções à sua linguagem hospedeira e é executável no mesmo ambiente de execução de programas escritos puramente em Java. Tal compatibilidade é alcançada pela transformação de construções específicas de AspectJ em correspondentes Java, e a posterior compilação dessas versões Java de construções AspectJ para código *bytecode*, executável em qualquer Máquina Virtual Java (JVM, do inglês *Java Virtual Machine*).

Costura de código (*weaving*) é a fase de introdução do comportamento de requisitos transversais no programa hospedeiro de uma linguagem orientada por aspectos implementada por ambientes de AOP. Em AspectJ, existem duas principais abordagens de implementação de costura de código: costura estática e costura dinâmica[1]. Costura estática é realizada como um passo de compilação, e modifica o código *bytecode* executável do programa base. Costura dinâmica é um processo realizado em tempo de execução, que modifica o comportamento do código carregado pela JVM para introduzir o comportamento de aspectos sem, no entanto, modificar o código arquivo do código executável original.

A costura estática de programas em AspectJ é um tema complexo, já que a escolha de construções Java para representar seus correspondentes em AspectJ pode influenciar a qualidade do código gerado. *Qualidade*, nesta discussão, é tratada como desempenho e tamanho do código gerado. Costura

[1] Traduções de *static weaving* e *dynamic weaving*, respectivamente

estática oferece mais possibilidades de otimização, pois torna-se possível realizar análises mais detalhadas do código, que, se fossem realizadas em tempo de execução, poderiam ser mais demoradas do que os ganhos obtidos como resultado. O principal objetivo deste trabalho é caracterizar as estratégias de costura estática implementada por compiladores da linguagem AspectJ, e, a partir dessa caracterização, identificar problemas existentes nessas estratégias e buscar soluções para esses problemas.

1.2 Motivação

Programação orientada por aspectos evoluiu, em menos de uma década, de um conceito teórico para um conjunto de linguagens de programação e arcabouços de amplo uso em desenvolvimento de sistemas comerciais [Lad03]. A tecnologia que suporta ferramentas AOP é intrinsecamente intrusiva, pois altera o comportamento do código-base de aplicações. A costura de código de aspectos em programas Java realizada por compiladores da linguagem AspectJ deve introduzir o comportamento esperado desses aspectos sem causar impactos de desempenho na execução de programas hospedeiros.

Nota-se, entretanto, que AOP e AspectJ são tecnologias recentes, e estratégias de compilação para programas escritos em AspectJ não são ainda tão consolidadas quanto técnicas mais tradicionais de geração de código, tais como análises de variáveis vivas e otimizações *peephole*. O estudo de estratégias de compilação adotadas por compiladores de AspectJ e a observação do código que eles produzem podem evidenciar áreas e casos especiais para melhorar a qualidade de código gerado para programas nessa linguagem.

1.3 Objetivos

O objetivo geral deste trabalho é estudar e buscar melhorias para as técnicas de costura de código existentes. Para que isso seja possível, tornam-se necessários também alguns objetivos específicos, tais como estudar as técnicas de costura estática usadas em diferentes compiladores de AspectJ, especificamente o *AspectJ Compiler*[2] (**ajc**) e o *AspectBench Compiler*[3] (**abc**).

A partir de problemas identificados nas técnicas de costura estudadas, como os mostrados na Seção 5, pretende-se desenvolver, implementar e avaliar algoritmos que os solucionem. Especificamente, busca-se, neste trabalho, solucionar o problema de redundância existente em código gerado pelos compiladores **ajc** e **abc**.

[2] http://www.eclipse.org/aspectj
[3] http://www.aspectbench.org

A última versão da linguagem AspectJ permite também costura dinâmica, como resultado da fusão entre essa linguagem e o arcabouço AspectWerkz. Outro objetivo deste trabalho consiste em estudar as técnicas de costura dinâmica aplicadas em AspectJ em busca de possíveis problemas e melhorias.

2 AspectJ

Programação Orientada por Aspectos (AOP) permite a implementação de programas complexos de forma modular, fornecendo maneiras de desenvolver diferentes requisitos independentemente como aspectos desses programas. Aspectos são costurados no código base para prover um único programa com a funcionalidade desejada cujo código é, não obstante, modular, fácil de manter e compreender. Deve-se observar que o produto final dessa metodologia de desenvolvimento é um programa com requisitos entrelaçados, porém fora do alcance do programador, que tem acesso a código modular e de fácil leitura e manutenção.

Um "mecanismo" que implemente AOP deve ser considerado como uma linguagem ou arcabouço que provê ferramentas para capturar pontos de entrelaçamento em um programa base e costura código de aspectos nesses pontos. AspectJ é uma extensão da linguagem Java que permite que programadores introduzam código de aspectos em pontos definidos de um programa base Java, tais como antes ou após chamadas de métodos.

Algumas definições são necessárias para a compreensão de construções de AspectJ; as definições descritas neste trabalho e algumas outras são detalhadas em [Lad03, GL03]. Na terminologia comum de AspectJ[4], um ponto de junção (*join point*) é um ponto enumerável na execução do programa, tal como uma chamada de método ou acesso ao valor de algum campo de uma classe. Conjuntos de junção (*pointcuts*) são expressões que selecionam pontos de junção, e que podem capturar seu contexto, como o argumento ou objeto alvo de uma chamada de método. Um adendo (*advice*), ou comportemento transversal, é uma porção de código a ser executada quando o ponto de junção selecionado por um conjunto de junção for alcançado. *Aspectos* são construções de alto nível, similares a classes Java, onde vários conjuntos de junção e adendos que implementam um dado requisito transversal podem ser definidos. Aspectos podem também definir seus próprios atributos e métodos, incluir novos métodos e atributos e alterar a hierarquia de classes do programa base; essa alteração do código do programa base é chamada de transversalidade estática (*static crosscutting*).

[4] Uma tabela completa das traduções para os português dos termos comuns relacionados a AOP podem ser encontradas no Apêndice A

Como AspectJ é uma extensão de Java, seu "mecanismo" gera código Java a partir de aspectos e adendos, e inclui chamadas para código de aspectos nos locais do programa base em que pontos de junção são selecionados. Esse processo, chamado de combinação ou costura (*weaving*), gera código *bytecode* como resultado, e portanto pode ser executado em JVMs comuns. Esse processo de costura é chamado de *costura estática*, já que é realizado em tempo de compilação e altera o código objeto do programa base. *Overheads* causados pelo código necessário à implementação da costura de código são discutidos na Seção 4.

Extensão de uma linguagem, entretanto, não deve ser considerada a única forma de implementar AOP. Outro exemplo bem sucedido de implementação de AOP é AspectWerkz [Bón04], que também possui Java como ambiente hospedeiro, porém provê uma filosofia de AOP puramente implementada em Java, ao invés de uma extensão de linguagem. Em AspectWerkz, o combinador (*weaver*) opera em tempo de execução na JVM, e aspectos são implementados como classes Java comuns. Conjuntos de junção são descritos por anotações no código fonte, e a estratégia de costura usa um arquivo descritor XML ou anotações no código para identificar quais classes são aspectos. Esse processo de costura dinâmica tem a vantagem de que o código base existente não precisa ser recompilado para incluir aspectos; em verdade, não é necessário um compilador adicional.

3 Compiladores de AspectJ

As definições informais de pontos de junção e adendos fornecidas em [Lad03, GL03] e descritas brevemente na Seção 2 são suficientes como uma visão geral de como um "mecanismo" de AOP tal como AspectJ pode ser usado. Entretanto, para determinar como implementar tal mecanismo, podem-se tornar necessárias as especificações formais das semânticas desses conceitos, como as encontradas em [WKD01]. Mais importante, o *modelo de compilação* fornecido em [MKD03] serve como guia para a compreensão de como essas características podem ser implementadas em um compilador de AOP.

Para descrever como construções de AspectJ podem ser costuradas, dois compiladores foram estudados: **ajc** e **abc**. O *AspectJ Compiler*, **ajc**, é o compilador "oficial" da linguagem AspectJ. Ele é baseado no compilador da linguagem Java do projeto Eclipse[5], e provê compilação incremental, que permite que a IDE Eclipse compile parcialmente o código fonte enquanto ele é escrito pelo programador, e continue essa compilação à medida que o código é completado.

[5] http://www.eclipse.org

O *AspectBench Compiler*, **abc**, foi desenvolvido para resolver problemas de desempenho causados pela costura de código do **ajc**, identificados em [DGH+04]. Esse compilador é também um interessante laboratório para experimentos com a linguagem AspectJ e técnicas de costura para suas construções. Seu combinador difere do implementado no **ajc** principalmente nos correspondentes em Java escolhidos para representar construções de AspectJ no código costurado, especialmente em adendos de contorno (*around advices*) e do tipo *cflow*, descritos na Seção 4.

4 Costura de Adendos

O foco deste trabalho está nas estratégias de costura de compiladores de AspectJ. Este estudo é baseado em código descompilado[6] a partir de arquivos de classes gerados tanto por **ajc** quanto por **abc**. Tal decisão foi baseada em questões de legibilidade: o código de pilha e as referências em *byte* de arquivos *bytecode* [LY99] podem ser difíceis de acompanhar durante a tentativa de compreender um programa, e, assim, reconstruir código Java a partir de *bytecode* simplifica a tarefa de compreender as estratégias de costura aplicadas pelos compiladores **ajc** e **abc**. As descrições de estratégias de costura presentes neste texto são baseadas em trabalhos escritos pelos projetistas dos compiladores **ajc** e **abc** [HH04, Kuz04, ACH+05b], além da análise de código gerado por eles.

Uma fase inicial do *frontend* do compilador, chamada casamento (*matching*), é responsável por identificar pontos no código do programa base em Java que casam com os conjuntos de junção do programa. Esses são os pontos de junção em que adendos se aplicam. Tais pontos também são chamados de pontos de hachura (*shadows points*) no contexto de adendos de contorno.

Aspectos são transformados em classes Java, e, como corpo de adendos é em geral composto de código Java[7], eles são transformados em métodos. O local onde esses métodos são inseridos pode variar dependendo da estrutura do programa e da técnica de costura adotada pelo compilador. Com a informação reunida na fase de casamento, o combinador insere chamadas para o método que implementa um adendo nos locais apropriados. Código de chamada de adendos pode incluir *resíduo dinâmico*, que consiste de código de testes que devem ser realizados em tempo de execução. Resíduo dinâmico pode resultar de cláusulas *if* em expressões de conjuntos de junção ou operações de teste para implementação de *cflow*.

[6] usando o descompilador Dava disponível como parte do arcabouço Soot [VGH+00]

[7] exceto quando são usadas a API de reflexão computacional de AspectJ e a instrução *proceed*, explicada adiante

Como sugere o nome, adendos anteriores (*before advices*) devem ser executados antes de um ponto de junção. Conjuntos de junção podem selecionar a chamada ou a execução de métodos, por meio das cláusulas *call* e *execution*, respectivamente. Em conjuntos de junção de execução, uma chamada ao método de adendo é incluído no início do método selecionado pela expressão do conjunto de junção, enquanto nos de chamada o método de adendo é ativado no objeto responsável pela chamada, logo antes do ponto de junção casado.

Adendos que se aplicam após um ponto de junção, chamados de adendos posteriores (*after advices*), possuem a mesma distinção quanto a conjuntos de junção com cláusulas de chamada e de execução. Esses adendos, no entanto, possuem uma peculiaridade: sua aplicação pode ser restrita a execuções de métodos que retornem (*after returning*) ou que lancem uma exceção (*after throwing*).

Se um adendo do tipo *after returning* se aplica a um conjunto de junção de chamada, o comportamento transversal é invocado após o método selecionado; se ele se aplica a um conjunto de junção de execução, o adendo é chamado ao final do método, antes da sua instrução de retorno. Caso várias instruções de retorno existam no corpo de um método, todas elas são substituídas por atribuições a uma variável temporária, e essa variável é retornada apenas *no final* do método, de modo que apenas uma chamada ao adendo posterior seja inserida no método. As mesmas idéias se aplicam para adendos do tipo *after throwing*, exceto pelo fato de que eles só podem ser chamados quando uma exceção ocorrer; comportamentos transversais posteriores sem restrições de aplicação são chamados de *after finally*, e se aplicam em ambos os casos.

Compotamentos transversais de contorno substituem a execução dos pontos de junção aos quais se aplicam. Eles devem ter o mesmo tipo de retorno dos pontos que selecionam. A maior dificuldade na costura desses adendos está relacionada à instrução *proceed* que pode aparecer em seu corpo: uma vez chamada, essa instrução deve executar o ponto de junção original. Como o mesmo adendo pode ser aplicado a múltiplos locais do código, o "código original" pode ser um de muitos pontos de junção do programa. Os compiladores **ajc** e **abc** adotam diferente estratégias na costura de comportamentos transversais desse tipo, que são descritas nas Seções 4.1 e 4.2, respectivamente. A costura de conjuntos de junção do tipo *cflow* é descrita na Seção 4.4.

Para simplificar a explicação das técnicas de costura utilizadas pelos compiladores **ajc** e **abc**, um exemplo simples de aplicação de adendos de contorno é analisado na Seção 4.5.

4.1 Adendos de Contorno no Compilador *ajc*

A estratégia para costurar comportamentos transversais de contorno adotada pelo compilador **ajc** é brevemente descrita em [HH04] e detalhada em trabalhos do grupo responsável pelo compilador **abc** [DGH+04, Kuz04, ACH+05a, ACH+05b]. O código do ponto de junção ao qual um adendo desse tipo se aplica é chamado de ponto de hachura (*shadow point*). Como a semântica da instrução *proceed* é executar o código da hachura original, esses fragmentos de código são extraídos para métodos, que são chamados de métodos de implementação de hachuras (*shadow methods*). Quaisquer variáveis existentes no contexto original do ponto de hachura são passados como parâmetros para o método correspondente a ele.

Em adendos de contorno onde uma instrução *proceed* é aninhada em uma classe anônima criada no corpo do adendo, o combinador substitui o código do ponto de hachura original pela instanciação de uma implementação do tipo `AroundClosure`. Como descrito em [HH04], essa interface de *closure* é simplesmente um local de implementação de código de hachura.

O objeto *closure* contém um método `run` aonde o código de hachura original é implementado, de forma que possa ser chamado em um ponto posterior da execução do programa. O corpo do adendo é implementado como um método da classe que implementa o aspecto ao qual ele pertence, e o objeto *closure* é passado como um argumento para esse método. Polimorfismo é usado para chamar o ponto de hachura apropriado no método que implementa o adendo, por meio do objeto *closure* fornecido como parâmetro.

Quando um objeto *closure* não é necessário, ou seja, quando não ocorrem hachuras do adendo no corpo de classes anônimas, a costura de adendos de contorno é mais simples: pontos de hachura são extraídos para métodos estáticos das classes em que originalmente apareciam. Para cada hachura de um adendo de contorno em uma classe A, por exemplo, um método com o corpo desse adendo é implementado nessa classe, onde a instrução *proceed* é substituída por uma chamada ao método daquela hachura.

4.2 Adendos de Contorno no Compilador *abc*

Kuzins, em [Kuz04], detalha a estrutura usada para implementar costura de adendos de contorno no compilador **abc**. *Benchmarks* apresentados mostram que essa estratégia gera código que executa mais rápido que o produzido pelo compilador **ajc**. A estratégia de costura apresentada nesta seção evita completamente a criação de objetos de *closure*, já que os *benchmarks* apresentados em [DGH+04] mostram que instanciação de *closures* pode causar um grande *overhead* em programas em que um adendo de contorno se aplica

a muitos pontos de junção.

Cada hachura de um adendo de contorno, nessa abordagem, é rotulada com um identificador inteiro, chamado *shadowID*, e cada classe onde hachuras de um adendo de contorno aparecem é também rotulada com um identificador chamado *classID*. Todas as hachuras de um dado adendo na mesma classe são extraídos para um método estático dessa classe. Se todas as hachuras de um adendo de contorno estiverem na mesma classe, a implementação do método correspondente a esse adendo é colocada nesta classe; caso contrário, ele aparece na classe do aspecto. A instrução *proceed* é então substituída por uma seleção de dois níveis: no método do adendo, a classe que contém a hachura apropriada é primeiramente identificada pelo *classID*, e então, no método de implementação de hachuras dessa classe, a hachura específica para cada execução do adendo é selecionada pelo seu *shadowID*.

O par *classID*-*shadowID*, em versões anteriores do compilador **abc**, aparecia no código final gerado, e assim a seleção da classe e da hachura para uma dada instrução *proceed* era realizada em tempo de execução. Essa estratégia foi levada um passo adiante na Versão 1.1.0 do compilador **abc**, que, em alguns casos, realiza a seleção em tempo de execução, por um processo de *inlining*. Esse passo adicional na costura realizado pelo **abc** reduz ainda mais o custo adicional da execução da instrução *proceed*.

4.3 Passagem de Contexto

Tanto na estratégia de costura adotada pelo **ajc** quanto pelo **abc**, a descrição deste texto ignorou a passagem de contexto para o adendo e para o método que implementa a hachura. Contexto pode conter variáveis usadas pela hachura e também informações de reflexão computacional usadas no corpo do adendo. O programador pode também capturar contexto na expressão de um conjunto de junção e usá-lo na instrução *proceed*, e, portanto, ele deve ser passado para o método do adendo para permitir que o adendo e a hachura executem corretamente. Descrições mais detalhadas da passagem de contexto podem ser encontradas em [HH04, Kuz04].

4.4 Conjuntos de Junção do tipo *cflow*

No escopo de costura de adendos em AspectJ, é necessário também considerar os chamados conjuntos de junção dinâmicos, que são definidos em termos de outros conjuntos de junção e capturam o fluxo de controle dinâmico da execução do programa. AspectJ provê as cláusulas *cflow* e *cflowbelow* na definição de conjuntos de junção, que permitem que o programador capture pontos que ocorram sob o fluxo dinâmico de outros conjuntos de junção,

oferecendo assim maior poder de expressão do que os conjuntos de junção baseados em propriedades, tais como nomes de métodos e hierarquia de classes, discutidos até este ponto.

A semântica de compilação para conjuntos de junção do tipo *cflow* em [MKD03] indica que uma pilha deve ser criada para representar o estado de cada conjunto de junção desse tipo, e manipulada na entrada e saída de seus pontos de junção correspondentes. A implementação direta dessa idéia aparece nas primeiras versões do compilador **ajc** (até 1.2). Essa estratégia é discutida em detalhe no restante desta seção e em seguida apresentam-se melhorias propostas a partir dela.

Um conjunto de junção da forma **cflow**(p) seleciona pontos de junção que ocorrem sob o fluxo de controle do *pointcut* p, e podem também capturar seu contexto. Um exemplo desse tipo de conjunto de junção, dado em [ACH+05b], é

```
pointcut foobar():
  call(* foo()) && cflow(call(* bar(*)) && args(x))
```

que captura chamadas ao método `foo`, sem argumentos, que ocorram sob o fluxo de controle de uma chamada a `bar` com argumento x de qualquer tipo[8].

Na compilação dessas construções, uma pilha pode ser associada a cada conjunto de junção do tipo *cflow* do programa. Sempre que uma chamada é feita ao método `bar`, um novo elemento de estado é armazenado na sua pilha. Elementos dessa pilha são listas de variáveis de contexto; neste exemplo, o valor de x usado na chamada atual a `bar` é colocado no topo da pilha deste conjunto de junção. Quando o controle sai do método `bar`, o elemento do topo é retirado da pilha. Como variáveis de contexto só são capturadas em conjuntos de junção do tipo *cflow* que possuem a cláusula *args*, os elementos das pilhas de estado que representam conjuntos de junção sem essa cláusula são listas vazias.

Para verificar se o conjunto de junção `foobar` se aplica em chamadas a `foo`, o combinador insere código nessas chamadas para verificar se a pilha é não-vazia; se variáveis de contexto forem ligadas pelo conjunto de junção do tipo *cflow*, os respectivos valores são recuperados de sua pilha.

Apesar de essa estratégia parecer razoável à primeira vista, várias melhorias foram implementadas pelos projetistas do compilador **abc**, e algumas delas foram também usadas em versões mais recentes do **ajc** [ACH+05b]. Algumas dessas otimizações são simples, e implementadas com análises locais do código, enquanto outras requerem análises inter-procedurais do fluxo de execução do programa.

[8] leitores não habituados com a notação da linguagem AspectJ podem consultar [Lad03, GL03, KHH+01]

As otimizações intra-procedurais simples são usadas em casos especiais de aplicação de conjuntos de junção desse tipo. Quando uma expressão de *cflow* não liga variáveis de contexto, os elementos de sua pilha de controle são sempre listas vazias. Para evitar instanciar e empilhar listas vazias, o compilador **abc** implementa o controle de estado desses conjuntos de junção como variáveis inteiras, cujos valores indicam o número de chamadas ativas em seu fluxo. Verificar se um conjunto de junção p está sob **cflow**`(p)` consiste então em verificar o valor de uma variável inteira. Além disso, quando dois conjuntos de junção do tipo *cflow* são sintaticamente idênticos exceto pelas variáveis de contexto que ligam, eles podem ser unificados em uma única pilha que contém os valores de contexto necessários para ambos. Essas duas otimizações foram também implementadas na Versão 1.2.1 do compilador **ajc**.

Embora essas otimizações simples produzam algum ganho de desempenho, a principal melhoria proposta para a costura desse tipo de conjuntos de junção está no uso de análises inter-procedurais: é possível determinar estaticamente que alguns pontos no programa *nunca* podem estar no escopo dinâmico de um ponto de junção e que outros *sempre* estão nesse escopo. Como em alguns desses pontos o resultado do teste de estado do conjunto de junção *cflow* é sempre conhecido, o resíduo dinâmico desse teste pode ser eliminado do código gerado. Avgustinov et al., em [ACH+05b], afirmam que existem dois tipos de hachuras associados a cada coonjunto de junção *cflow* que causam *overhead* na execução do programa: uma hachura de atualização (*update shadow*) e uma hachura de consulta (*query shadow*).

Para um dado conjunto de junção `q &&` **cflow**`(p)`, hachuras de atualização são aquelas que casam com p, onde a estrutura de controle de estado associada ao conjunto de junção é manipulada – na entrada e saída de pontos selecionados por p. Hachuras de consulta são pontos q na execução do programa onde a "atividade" do conjunto de junção deve ser testada. Na expressão de *cflow* `foobar` da Página 11, hachuras de consulta são chamadas ao método `foo`. Nessas hachuras, é necessário inserir código para testar se a pilha de controle de estado associada a **cflow**`(p)` é vazia, ou se o valor da variável inteira associada é zero.

Se for estaticamente determinado que a pilha associada a um conjunto de junção do tipo *cflow* é sempre vazia em um ponto no programa, o resíduo dinâmico associado ao teste do *cflow* pode ser removido dessa hachura de consulta. Além disso, se a estrutura de estado associada a um conjunto de junção do tipo *cflow* nunca for consultada, todas as operações de atualização sobre esse *cflow* podem ser removidas das hachuras de atualização referentes a esse conjunto de junção.

Essa análise permite que o combinador elimine praticamente todos os

testes dinâmicos em tempo de compilação, produzindo considerável ganho de desempenho em programas que dependem de conjuntos de junção do tipo *cflow* [ACH+05b].

4.5 AspectJ Descompilado

O exemplo apresentado nesta seção é usado para ajudar a compreensão de estratégias de costura adotadas pelos compiladores **ajc** e **abc**, e a Seção 5 retorna a ele para ilustrar problemas no código gerado por esses compiladores. Deve-se notar que código descompilado contém nomes de variáveis e métodos gerados automaticamente e pode ser de difícil leitura; as porções importantes dessas listagens são descritas em detalhe. Listagens de código descompilado contidas neste texto são chamadas de *esqueletos* porque parte do código é omitida e alguns nomes de métodos alterados com a intenção de aumentar a legibilidade.

Considere o código das Listagens 1 e 2. A classe `Point` implementa pontos com coordenadas inteiras às quais podem ser atribuídas valores absolutos, por meio do método `setPosition(int,int)`, ou com deslocamentos, por meio de `moveBy(int,int)`. `Main` é uma classe de teste que cria uma instância de `Point` e realiza algumas operações sobre essa instância.

Listagem 1: Implementação de pontos com coordenadas inteiras.

```
1  public class Point {
2      private int x;
3      private int y;
4
5      public Point(int x, int y) {
6          setPosition(x,y);
7      }
8      public int getX() { return x; }
9      public int getY() { return y; }
10     public void setPosition(int x, int y) {
11         this.x = x;
12         this.y = y;
13     }
14     public void moveBy(int dx, int dy) {
15         setPosition(this.x + dx, this.y + dy);
16     }
17     public String toString() {
18         return "(" + x + "," + y + ")";
19     }
20 }
```

Listagem 2: Classe de teste para uma instância de `Point`.

```
1  public class Main {
2
3      public static void main(String[] args) {
4          Point p = new Point(0,0);
5          p.moveBy(14,27);
6          System.out.println(p);
7          doSomething(p);
8      }
9      private static void doSomething(Point p) {
10         p.setPosition(-5,5);
11         System.out.println(p);
12         p.moveBy(40,100);
13         System.out.println(p);
14     }
15 }
```

A saída esperada para uma execução de `Main` é

(14,27)
(-5,5)
(35,105)

Suponha, agora, que um novo requisito seja incluído na especificação do programa: coordenadas de pontos não podem ser negativas. Usando AOP, é possível separar a implementação desse requisito, que abrange toda a aplicação, em um aspecto, ao invés de alterar os métodos afetados na classe `Point`[9]. Escreve-se, assim, o aspecto da Listagem 3.

`RangeCheckAspect` define um conjunto de junção chamado `setMethod` para capturar chamadas ao método `Point.setPosition(int,int)`. No programa base definido acima, o *pointcut* `setMethod` seleciona pontos de junção nas Linhas 6 e 15 de `Point` e na Linha 10 de `Main`. Quando o programa base é compilado junto com esse aspecto, sua saída se torna

(14,27)
(0,5)
(40,105)

[9] apesar de a qualidade dessa solução ser questionável, ela cumpre o propósito deste exemplo e evidencia problemas nas estratégias de costura de adendos de contorno

Listagem 3: Um aspecto para garantir que coordenadas de pontos não são negativas.

```
1  public aspect RangeCheckAspect {
2      pointcut setMethod(int x, int y)
3          : call(* Point.setPosition(int,int))
4            && args(x,y);
5
6      void around(int x, int y) : setMethod(x,y) {
7          if (x < 0) x = 0;
8          if (y < 0) y = 0;
9
10         proceed(x,y);
11     }
12 }
```

O adendo de contorno definido em `RangeCheckAspect` garante que a restrição de "coordenadas não-negativas" seja respeitada, capturando e manipulando os argumentos para cada chamada a `setPosition(int,int)`. A seguir, serão descritas as estratégias dos compiladores **ajc** e **abc** aplicadas a este exemplo.

O que o *ajc* faz

Pela estratégia de costura de adendos de contorno descrita na Seção 4.1, um método é criado para implementar o corpo do adendo e outro para a hachura. Para o código do programa base deste exemplo, ambos os métodos são colocados na classe onde a hachura aparece.

O esqueleto do código descompilado para a classe `Main` da Listagem 4 mostra o resultado da costura do aspecto `RangeCheckAspect` na classe `Main` original. O método implementado nas Linhas 25-39 é o corpo do *advice*. Nesse adendo costurado, entretanto, a instrução *proceed* foi substituída pela chamada, na Linha 38, pelo método para onde a hachura foi extraída, `setPosition_aroundBody0`, implementado nas Linhas 20-23. A hachura que existia no método `doSomething` foi substituída pela chamada ao método do adendo das Linhas 12-14.

Listagem 4: Esqueleto do código descompilado da classe `Main`, a partir do *bytecode* gerado pelo **ajc**.

```
1  public class Main
2  {
3
4      /* ... */
5
6      private static void doSomething(Point r0) {
7          int i0;
8          byte b1;
9
10         i0 = 5;
11         b1 = (byte) -5;
12         Main.setPosition_aroundBody1$advice
13             (r0, b1, i0, RangeCheckAspect.aspectOf(),
14              b1, i0, null);
15         System.out.println(r0);
16         r0.moveBy(40, 100);
17         System.out.println(r0);
18     }
19
20     private static final void setPosition_aroundBody0
21             (Point r0, int i0, int i1) {
22         r0.setPosition(i0, i1);
23     }
24
25     private static final void setPosition_aroundBody1$advice
26             (Point r0, int i0, int i1,
27             RangeCheckAspect r1, int i2, int i3,
28             org.aspectj.runtime.internal.AroundClosure r2) {
29
30         if (i2 < 0) {
31             i2 = 0;
32         }
33
34         if (i3 < 0) {
35             i3 = 0;
36         }
37
38         Main.setPosition_aroundBody0(r0, i2, i3);
39     }
40 }
```

Apesar de, para este programa, o requisito transversal ser costurado nas classes do programa, **ajc** é um compilador incremental. Assim, ele sempre gera código assumindo que mais pontos de junção para adendos existentes

podem aparecer em futuras compilações. O esqueleto de código mostrado na Listagem 5 contém a implementação por meio de *closures* para o *around advice* de `RangeCheckAspect`, que também é gerado pelo **ajc**, apesar de nunca ser usado neste exemplo. A implementação do padrão de projeto *Singleton* [GHJV95] e outros detalhes foram omitidos nessa listagem; o símbolo '_' substitui grandes porções de nomes de métodos gerados automaticamente.

Ainda na Listagem 5, note que o método que implementa o adendo, `ajc$around_`, chama `ajc_proceed` na Linha 17 e passa para ele o objeto `AroundClosure` recebido como parâmetro. Esse objeto é usado pelo método *proceed* na Linha 28, onde o método `run` é chamado para executar a hachura. Esse objeto é instanciado, quando necessário, no local original do código base onde a hachura apareceu, e o código da hachura é extraído para o método `run` dessa instância anônima da interface `AroundClosure`. O código descompilado para a classe `Point` não é mostrado nesta seção porque o código do aspecto costurado nessa classe é muito similar ao costurado em `Main`. As alterações causadas pela costura na classe `Main` são a inclusão de métodos que implementam o adendo e a extração da hachura para um método próprio.

Listagem 5: Esqueleto do código descompilado do aspecto `RangeCheckAspect`, a partir do *bytecode* gerado pelo **ajc**.

```
1  public class RangeCheckAspect
2  {
3      /* ... */
4
5      public void ajc$around_
6              (int i0, int i1,
7               org.aspectj.runtime.internal.AroundClosure r1)
8               throws java.lang.Throwable {
9
10         if (i0 < 0) {
11             i0 = 0;
12         }
13         if (i1 < 0) {
14             i1 = 0;
15         }
16
17         RangeCheckAspect.ajc_proceed(i0, i1, r1);
18     }
19     static void ajc_proceed
20             (int i0, int i1,
21              org.aspectj.runtime.internal.AroundClosure r0)
22              throws java.lang.Throwable {
23         java.lang.Object[] $r1;
24
25         $r1 = new Object[2];
```

```
26          $r1[0] = Conversions.intObject(i0);
27          $r1[1] = Conversions.intObject(i1);
28          Conversions.voidValue(r0.run($r1));
29      }
30
31      /* ... */
32  }
```

É importante ressaltar alguns problemas que aparecem nas listagens mostradas nesta seção. O compilador **ajc**, durante a compilação incremental, produz métodos e variáveis que podem ser necessários em futuras compilações, mas que podem não ser usados de fato no programa. Esse é o caso dos métodos `ajc$around_` e `ajc_proceed` da classe `RangeCheckAspect`, mostrada na Listagem 5. Eles são gerados para a costura de aplicações em classes anônimas do adendo de contorno definido no programa, que, como pode ser observado no código das Listagens 1 e 2, não ocorrem. O mesmo acontece com as variáveis de contexto passadas para adendos. Como exemplo, observe o corpo do método `setPosition_aroundBody1$advice` definido nas Linhas 25-39. Os parâmetros `i0`, `i1`, `r1` e `r2` nunca são usados.

O que o *abc* faz

O esqueleto de código desta seção foi descompilado a partir do gerado pela Versão 1.1.0 do compilador **abc**. A seleção da hachura apropriada para cada execução da instrução *proceed* é realizada em tempo de compilação, e o código final gerado não contém instruções para essa tarefa. Novamente, o código descompilado para a classe `Point` não é mostrado nesta seção, já que o código nele costurado é muito similar ao que aparece em `Main`.

No esqueleto de código da Listagem 6, não há implementação do adendo. A instância *singleton* de `RangeCheckAspect` é obtida na Linha 7, e o adendo de contorno aplicado nesse ponto é chamado na Linha 8; o adendo é implementado na classe que implementa o aspecto ao qual ele pertence.

Listagem 6: Esqueleto do código descompilado da classe `Main`, a partir do *bytecode* gerado pelo **abc**.

```
1  public class Main
2  {
3      /* ... */
4
5      private static void doSomething(Point r0)
6      {
7          RangeCheckAspect.aspectOf();
8          RangeCheckAspect.inline$2$around$3(r0);
9          System.out.println(r0);
10         r0.moveBy(40, 100);
11         System.out.println(r0);
12     }
13     /* ... */
14 }
```

Listagem 7: Esqueleto do código descompilado do aspecto `RangeCheckAspect`, a partir do *bytecode* gerado pelo **abc**.

```
1  public class RangeCheckAspect
2  {
3      /* ... */
4      public static final void inline$2$around$3
5              (Point r0) {
6          r0.setPosition(0, 5);
7      }
8      public static final void inline$7$around$3
9              (int i0, int i1, int i2,
10              int i3, Point r0) {
11         int i4, i5;
12         i4 = i0;
13         i5 = i1;
14         if (i0 < 0) {
15             i4 = 0;
16         }
17         if (i1 < 0) {
18             i5 = 0;
19         }
20         r0.setPosition(i4, i5);
21     }
22     /* ... */
23 }
```

O código descompilado para `RangeCheckAspect` a partir do código gerado pelo **abc** é mostrado na Listagem 7. O método implementado nas Linhas 4-7 é uma aplicação especial do adendo que ocorre em `Main.doSomething`. Observe na Listagem 2 da Página 14 que há uma chamada a `Point.doSomething` com argumentos (-5,5). Durante a costura, o compilador **abc** determina que o primeiro parâmetro é sempre transformado em `0` pelo adendo de contorno definido em `RangeCheckAspect`, e assim elimina o teste em tempo de execução para essa aplicação do adendo criando a sua implementação especial `inline$2$around$3`.

A implementação do método de adendo `inline$7$around$3` das Linhas 8-21 é aplicada a outros pontos do código base, onde o valor dos parâmetros de chamadas a `Point.setPosition` não é conhecido em tempo de execução. A instrução *proceed*, nesse método, foi substituída pela chamada à hachura na Linha 20.

Note que não há objetos de *closure* neste programa costurado, e essa ausência é um dos principais ganhos de desempenho alcançados pela estratégia de costura do compilador **abc**.

5 Problemas identificados

Como descrito na Seção 4, instruções *proceed* no corpo de um adendo de contorno devem executar o código da hachura apropriada para cada ponto de junção selecionado por esse adendo. Para mostrar problemas identificados nas estratégias de costura adotadas pelos compiladores **ajc** e **abc**, essa seção revê o exemplo da Seção 4.5.

O combinador do compilador **ajc** insere uma implementação do corpo do adendo nas classes `Point` e `Main`. Cada hachura desse adendo é extraída para um método na classe onde ela aparece.

A estratégia de *inlining* adotada na versão mais recente do compilador **abc** cria uma implementação do adendo para cada hachura na classe correspondente ao aspecto em que esse adendo foi definido, e substitui a instrução *proceed* dessas implementações por chamadas ao método que o ponto de junção original chama, que, para esse exemplo, é `Point.setPosition`.

Note uma característica interessante deste programa: todas as hachuras do adendo de contorno existente são chamadas para o mesmo método, e as únicas distinções entre eles são o local da chamada e os argumentos passados. Informações de contexto usadas na hachura são passadas como parâmetros para os métodos que implementam o adendo e a hachura, e, portanto, o código extraído para os métodos que implementam hachuras é exatamente o mesmo para todas as hachuras. Tanto o compilador **ajc** quanto o **abc**

ignoram essa repetição de código de hachuras idênticas, que, apesar de não parecer impactante para este exemplo simples, pode replicar grandes porções de código em adendos de contorno mais complexos que se aplicam a vários pontos de junção em um mesmo programa.

Para ilustrar esse problema, o esqueleto do código descompilado a partir do *bytecode* gerado pelo **ajc** para a classe `Point` é mostrado na Listagem 8. Nessa classe, existem duas hachuras para o adendo de contorno de `RangeCheckAspect`, e então o combinador desse compilador gera dois métodos que implementam hachuras, `setPosition_aroundBody0` e `setPosition_aroundBody2`. Duas implementações do corpo do adendo também são geradas pelo combinador do **ajc**, nos métodos `setPosition_aroundBody1$advice` e `setPosition_aroundBody3$advice`, que chamam suas respectivas hachuras. Note que os conteúdos desses dois pares de métodos são *idênticos*, e a única distinção entre eles está nos parâmetros que eles recebem. Essa similaridade é uma evidência de que métodos gerados pelo combinador para implementar hachuras e adendos podem ser unidos para produzir um código menor como resultado do processo de costura.

Listagem 8: Esqueleto do código descompilado da classe `Point`, a partir do *bytecode* gerado pelo **ajc**.

```
1  public class Point {
2      /* ... */
3      private static final void setPosition_aroundBody0
4              (Point r0, Point r1, int i0, int i1) {
5          r1.setPosition(i0, i1);
6      }
7      private static final void setPosition_aroundBody1$advice
8              (Point r0, Point r1, int i0, int i1,
9               RangeCheckAspect r2, int i2, int i3,
10              org.aspectj.runtime.internal.AroundClosure r3) {
11         if (i2 < 0) {
12             i2 = 0;
13         }
14         if (i3 < 0) {
15             i3 = 0;
16         }
17         Point.setPosition_aroundBody0(r0, r1, i2, i3);
18     }
19     private static final void setPosition_aroundBody2
20             (Point r0, Point r1, int i0, int i1) {
21         r1.setPosition(i0, i1);
22     }
23     private static final void setPosition_aroundBody3$advice
24             (Point r0, Point r1, int i0, int i1,
25              RangeCheckAspect r2, int i2, int i3,
```

```
26                org.aspectj.runtime.internal.AroundClosure r3) {
27            if (i2 < 0) {
28                i2 = 0;
29            }
30            if (i3 < 0) {
31                i3 = 0;
32            }
33            Point.setPosition_aroundBody2(r0, r1, i2, i3);
34        }
35    }
```

O mesmo problema aparece em código gerado pelo compilador **abc**. Na Seção 4.5, a Listagem 7 mostrou um esqueleto do código costurado para o aspecto `RangeCheckAspect`, descompilado a partir do *bytecode* gerado pelo **abc**. Nessa listagem, entretanto, omitiu-se uma implementação do adendo; o programa base constituído das classes `Point` e `Main` contém exatamente três hachuras para o adendo de contorno de `RangeCheckAspect`. Todas as implementações de *advice* do aspecto `RangeCheckAspect` são mostradas na Listagem 9.

O método `inline$2$around$3` corresponde a uma aplicação especial do adendo definido em `RangeCheckAspect` que resulta de um algoritmo da propagação de uma constante usada como argumento no código da hachura desse adendo existente na classe `Main`. Na Listagem 1 da Página 13 observa-se que o código da classe `Point` contém duas chamadas ao método `setPosition` e, portanto, duas hachuras do adendo de contorno de `RangeCheckAspect`. Assim como **ajc**, o compilador **abc** não unifica os métodos que implementam o adendo criados na costura dessas hachuras, produzindo os métodos com conteúdos idênticos `inline$7$around$3` e `inline$8$around$3`.

Listagem 9: Esqueleto do código descompilado do aspecto `RangeCheckAspect`, a partir do *bytecode* gerado pelo **abc**.

```
1   public class RangeCheckAspect {
2       /* ... */
3       public static final void inline$2$around$3
4               (Point r0) {
5           r0.setPosition(0, 5);
6       }
7       public static final void inline$7$around$3
8               (int i0, int i1, int i2,
9                int i3, Point r0) {
10          int i4, i5;
11          i4 = i0;
12          i5 = i1;
```

```
13          if (i0 < 0) {
14              i4 = 0;
15          }
16          if (i1 < 0) {
17              i5 = 0;
18          }
19          r0.setPosition(i4, i5);
20      }
21      public static final void inline$8$around$3
22              (int i0, int i1,
23               int i2, int i3,
24               Point r0) {
25          int i4, i5;
26
27          i4 = i0;
28          i5 = i1;
29
30          if (i0 < 0) {
31              i4 = 0;
32          }
33          if (i1 < 0) {
34              i5 = 0;
35          }
36          r0.setPosition(i4, i5);
37      }
38  }
```

Outro problema que aparece em código costurado por esses compiladores é a repetição de variáveis de contexto. Pela observação de código descompilado, parece claro que a passagem de contexto capturado pelo programador em expressões de conjuntos de junção e de outras informações de contexto necessárias para a execução da hachura são etapas separadas. Por exemplo, a expressão do conjunto de junção que captura a chamada a `setPosition` da Linha 6 da classe `Point`, definida na Listagem 1, captura as variáveis x e y. No momento de extração dessa hachura para um método, o contexto disponível no local de onde ele é removido constitui de x, y e a instância da classe `Point` (**this**) sobre a qual o método é chamado. Os métodos criados pelo combinador recebem os valores x e y duas vezes, como resultado da captura do contexto definido pela expressão do conjunto de junção e dos dados disponíveis no local de onde o código da hachura foi extraído.

6 Solução proposta

O problema de repetição de código de adendos e hachuras, contextualizado nas seções anteriores, torna o código gerado por compiladores da linguagem AspectJ maiores, e que, conseqüentemente, requerem mais tempo para serem carregados por JVMs que os executem.

A solução proposta para este problema consiste em unificar, durante o processo de costura, hachuras e métodos de implementação de adendos que são similares e uni-los em uma única implementação. Essa união deve manter o comportamento esperado do programa, melhorando a qualidade do código final gerado por compiladores da linguagem AspectJ.

Um raciocínio geral para o algoritmo de unificação de hachuras e adendos produz os seguintes passos em alto nível, que devem ser refinados na criação de uma solução concreta para o problema:

1. durante a costura, identificar todos os pontos de aplicação de um dado adendo de contorno, produzindo um conjunto sh de hachuras desse adendo;

2. determinar conjuntos $ish \subseteq sh$ de *hachuras idênticas*, cujos elementos são hachuras que consistem de chamadas ao mesmo método;

3. para cada conjunto de hachuras $s \in ish$, gerar código de implementação do adendo e da hachura s.

A implementação deste algoritmo depende da análise de diversas possibilidades e restrições. A solução proposta é um passo de compilação de programas na linguagem AspectJ, e, portanto, sua implementação deve ser integrada a um compilador dessa linguagem. O desenvolvimento de um compilador completo para a linguagem, entretanto, é uma tarefa despendiosa, e que foge ao escopo deste trabalho, de forma que pretende-se integrar a implementação do algoritmo de união de pontos de junção a compiladores existentes da linguagem.

Integrar o algoritmo nos compiladores **ajc** e **abc**, no entanto, requer um profundo conhecimento de suas arquiteturas. Embora a integração não altere o algoritmo em alto nível, os dois compiladores operam sobre diferentes estruturas de dados para a representação intermediária de código durante a costura.

O compilador **ajc** opera diretamente sobre código *bytecode* executável. A primeira etapa de compilação produz código *bytecode* a partir do código-fonte AspectJ e do código Java independentemente, incluindo anotações no

código gerado para indicar as ligações entre as construções Java e os aspectos do programa. A segunda etapa realiza as transformações indicadas nos atributos do *bytecode* sobre o código gerado, introduzindo no programa os requisitos tranversais implementados nos aspectos. *Bytecode* é um código de pilha [LY99], e por isso a modificação desse código durante a combinação de adendos no programa deve se preocupar em recuperar variáveis locais da pilha implícita de execução e em manter o estado da pilha após a execução do adendo, de forma que o código do programa continue a produzir o efeito desejado.

A dificuldade na manipulação de código *bytecode* necessária para a implementação da solução proposta é atenuada por meio de ferramentas de análise de *bytecode*, tais como a *Byte Code Engineering Library (BCEL)*[10]. Nessa biblioteca, classes são representadas por objetos que contém informações sobre métodos, atributos e instruções de código *bytecode* carregados. Conectar o uso da biblioteca BCEL ao compilador **ajc** é mais uma tarefa necessária para a implementação do algoritmo proposto nesse compilador.

A compilação de programas AspectJ no compilador **abc** também possui uma etapa de geração de representação intermediária acrescida de informações sobre aspectos e uma etapa de costura separada. Ao contrário do que ocorre no compilador **ajc** no entanto, código *bytecode* não é usado diretamente na costura de adendos no compilador **abc**. Ao contrário, esse compilador utiliza uma representação de mais alto nível de código executável *bytecode*, chamada *Jimple* [ACH+05a]. Código Jimple é tipado e não possui pilhas implícitas, e, segundo os desenvolvedores do compilador **abc**, mais fácil de manipular durante a costura de código do que *bytecode* diretamente.

7 Metodologia

O primeiro passo no desenvolvimento de uma solução viável para este problema na costura de programas em AspectJ consiste em resolver o problema *manualmente*. Essa "costura à mão" inicial ajudará a evidenciar passos e restrições importantes na união de hachuras para a criação de um algoritmo correto e abrangente que resolva o problema para casos comuns de aplicação de adendos de contorno.

Uma vez criado tal algoritmo, planeja-se implementá-lo nos compiladores **ajc** e **abc**, que são de código aberto. A escolha de implementar o algoritmo em compiladores existentes causa uma grande redução de escopo do trabalho necessário, já que outras etapas de compilação, como análise léxica e

[10] http://jakarta.apache.org/bcel

sintática, além da costura de outras construções de AspectJ, já estão implementadas. Essa abordagem permite concentrar o esforço de implementação deste trabalho no algoritmo proposto.

Existem duas validações principais a serem realizadas sobre o algoritmo desenvolvido:

1. correção: programas costurados com esse algoritmo devem comportar-se de acordo com a especificação da linguagem; e

2. qualidade e desempenho: como o algoritmo deste trabalho é proposto para melhorar a qualidade do código gerado na costura de programas em AspectJ, o código gerado por ele deve ser mais conciso e, portanto, de tempo de carga menor do que código gerado por meio das estratégias atualmente implementadas nos compiladores **ajc** e **abc**.

A medição dessas diretrizes de validação para o algoritmo proposto se dará através de um *benchmark* de aplicações de adendos de contorno. Para que o trabalho seja considerado bem sucedido, todos os testes propostos deverão compilar com sucesso, e será exigido também um ganho de desempenho, na média, em comparação com a execução dos mesmos programas compilados pelas estratégias originais de costura implementadas nos compiladores **ajc** e **abc**.

As principais dificuldades deste trabalho estão na implementação do algoritmo proposto em compiladores existentes. Será necessário estudar a arquitetura dos compiladores **ajc** e **abc**, com o objetivo de compreender as modificações necessárias para implementar a união de pontos de junção proposta. A integração entre o algoritmo desenvolvido e as estruturas de dados e etapas de compilação implementadas nesses compiladores é um processo importante deste trabalho, e que demandará grande parte do tempo disponível, como pode ser observado no cronograma da Seção 9. Além disso, a formação de um *benchmark* para validação dos resultados obtidos é também uma tarefa dispendiosa, já que requer a identificação de casos comuns de utilização de programação orientada por aspectos, e principalmente de adendos de contorno, para verificar o impacto causado pelo algoritmo proposto no desempenho de programas em AspectJ.

8 Contribuições Pretendidas

Com este trabalho, pretende-se oferecer as seguintes contribuições para a área de Linguagens de Programação:

1. caracterização detalhada das técnicas existentes de compilação de programas orientados por aspectos, especialmente na linguagem AspectJ;

2. avaliação do desempenho e qualidade do código gerado por compiladores de AspectJ;

3. identificação de possíveis problemas nessas técnicas e de áreas e casos especiais para os quais os compiladores de AspectJ podem ser melhorados;

4. formalização semântica das técnicas de compilação para a linguagem AspectJ existentes e do problema identificado; e

5. criação de algoritmos para melhorar a qualidade de código gerado por compiladores de linguagens orientadas por aspectos.

9 Cronograma

Janeiro de 2006:

- projeto do algoritmo de unificação de hachuras;
- aplicação manual do algoritmo a programas de teste e comparação do tempo de execução do código produzido por essa estratégia com os tempos de execução dos códigos gerados pelos compiladores **ajc** e **abc**;
- escrita de artigo para o X SBLP (Simpósio Brasileiro de Linguagens de Programação) descrevendo os resultados parciais obtidos na aplicação manual da técnica proposta.

Fevereiro:

- caracterização de estratégias de costura dinâmica para programas em AspectJ.

Março e Abril:

- estudo da arquitetura dos compiladores **ajc** e **abc** e identificação dos pontos de extensão a serem usados para a implementação do algoritmo proposto;
- implementação, nesses compiladores, do algoritmo proposto para união de pontos de junção.

Maio e Junho:

- construção de *benchmark* de aplicações de adendos de contorno que seja suficientemente representativo de casos comuns de aplicação dessas construções em programas AspectJ;
- aplicação do algoritmo implementado a esse conjunto de programas;
- identificação e correção de erros existentes na implementação ou no algoritmo.

Julho:

- escrita dos Capítulos 1 a 5 da dissertação;

Agosto:

- determinação de medidas e métodos de comparação entre a estratégia proposta e as implementadas pelos compiladores **ajc** e **abc**;
- aplicação e coleta de dados de testes comparativos entre as estratégias.

Setembro:

- avaliação dos resultados obtidos;
- escrita de artigo sobre os resultados obtidos.

Outubro e Novembro:

- escrita dos Capítulos 6 e 7 da dissertação;
- revisão do texto da dissertação.

Dezembro:

- defesa de dissertação.

10 Sumário da dissertação

1. Introdução

   1.1. Programação Orientada por Aspectos

   1.2. Abordagens de Implementação de AOP

11 Conclusão

O objetivo deste trabalho de dissertação é o desenvolvimento de técnicas de compilação que melhorem a qualidade do código gerado por compiladores de linguagens orientadas por aspectos por meio da unificação de pontos de aplicação de adendos. Nesta proposta, realiza-se uma breve revisão de literatura, na qual se busca evidenciar o contexto do problema e indicar o caminho de uma possível solução.

A unificação de hachuras produzirá, como resultado, código objeto menor e com menos redundância, o que, acredita-se, melhorará o desempenho de programas escritos na linguagem AspectJ que possuam muitas aplicações de adendos de contorno.

Referências

[ACH+05a] Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondrej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble, *abc: An Extensible AspectJ Compiler*, TAOSD'05 (2005).

[ACH+05b] ———, *Optimising AspectJ*, PLDI'05 (2005).

[Bón04] Jonas Bóner, *AspectWerkz – dynamic AOP for Java*, AOSD'04 (2004).

[DGH+04] Bruno Dufour, Christopher Goard, Laurie Hendren, et al., *Measuring the Dynamic Behaviour of AspectJ Programs*, OOPSLA'04 (2004).

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns – Elements of Reusable Object-oriented Software*, Addison-Wesley, 1995.

[GL03] Joseph D. Gradecki and Nicholas Lesiecki, *Mastering AspectJ*, John Wiley & Sons, Inc., 2003.

[HH04] Erik Hilsdale and Jim Hugunin, *Advice Weaving in AspectJ*, AOSD'04 (2004).

[KHH+01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold, *An Overview of AspectJ*, ECOOP '01 (2001).

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin, *Aspect-Oriented Programming*, ECOOP'97 (1997).

[Kuz04] Sascha Kuzins, *Efficient Implementation of Around-advice for the AspectBench Compiler*, Master's thesis, Oxford University, 2004.

[Lad03] Ramnivas Laddad, *AspectJ in Action*, Manning Publications Co., 2003.

[LY99] Tim Lindholm and Frank Yellin, *The Java Virtual Machine Specification*, second ed., Addison-Wesley Professional, 1999, Disponível em http://java.sun.com/docs/books/vmspec/index.html.

[MKD03] Hidehiko Masuhara, Gregor Kiczales, and Chris Dutchyn, *A Compilation and Optimization Model for Aspect-Oriented Programs*, Lecture Notes in Computer Science (2003).

[TBBV04] Fabio Tirelo, Roberto S. Bigonha, Mariza A. S. Bigonha, and Marco Túlio Oliveira Valente, *Desenvolvimento de Software Orientado por Aspectos*, XXIII Jornada de Atualização em Informática – JAI'04 (2004).

[VGH+00] Raja Vallee-Rai, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, Patrice Pominville, and Vijay Sundaresan, *Optimizing Java Bytecode Using the Soot Framework: Is It Feasible?*, Compiler Construction, 2000, pp. 18–34.

[WKD01] Mitchell Wand, Gregor Kiczales, and Christopher Dutchyn, *A Semantics for Advice and Dynamic Join Points in Aspect-Oriented Programming*, Lecture Notes in Computer Science (2001).

Apêndice

A Termos Relacionados a AspectJ

Tabela 1 mostra os termos em português adotados neste texto para a adaptação dos originais, em inglês. Porções entre '[' e ']' desses termos são normalmente omitidas no texto. Essa tradução é baseada em uma tabela de termos em português para Programação Orientada por Aspectos discutida durante o I Workshop Brasileiro de Software Orientado por Aspectos, em 2004[11], e em algumas traduções adotadas no tutorial apresentado por Fabio Tirelo durante o XXIV Congresso da Sociedade Brasileira de Computação [TBBV04].

Termo original	Traduções adotadas
Join point	Ponto de junção
Pointcut	Conjunto [de pontos] de junção
(before, after, around) advice	Adendo ou comportamento transversal (anterior, posterior, de contorno)
Aspect	Aspecto
Concern	Requisito, interesse
Crosscut	Atravessar, cruzar
Crosscutting concern	Requisito transversal
Weaving	Costura, combinação
Weaver	Combinador
Shadow point	Ponto de hachura

Tabela 1: Termos em português para Programação Orientada por Aspectos

[11] A tabela de traduções original pode ser encontrada em http://twiki.im.ufba.br/bin/view/WAsp/Termos