EDUARDO SANTOS CORDEIRO

OTIMIZAÇÕES NA COMPILAÇÃO DE ADENDOS DE CONTORNO EM PROGRAMAS ORIENTADOS POR ASPECTOS

Belo Horizonte, Minas Gerais
Fevereiro de 2007

Universidade Federal de Minas Gerais
Instituto de Ciências Exatas
Programa de Pós-Graduação em Ciência da Computação

OTIMIZAÇÕES NA COMPILAÇÃO DE ADENDOS DE CONTORNO EM PROGRAMAS ORIENTADOS POR ASPECTOS

Dissertação apresentada ao Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

EDUARDO SANTOS CORDEIRO

Belo Horizonte, Minas Gerais
Fevereiro de 2007

UNIVERSIDADE FEDERAL DE MINAS GERAIS

FOLHA DE APROVAÇÃO

Otimizações na Compilação de Adendos de Contorno em Programas Orientados por Aspectos

EDUARDO SANTOS CORDEIRO

Dissertação defendida e aprovada pela banca examinadora constituída por:

Roberto da Silva Bigonha – Orientador
Universidade Federal de Minas Gerais

Mariza Andrade da Silva Bigonha – Co-orientador
Universidade Federal de Minas Gerais

Belo Horizonte, Minas Gerais, Fevereiro de 2007

Resumo

Em menos de uma década, Programação Orientada por Aspectos (AOP) evoluiu de um conceito teórico para um conjunto de linguagens de programação e arcabouços de amplo uso no desenvolvimento de sistemas comerciais. A tecnologia que suporta ferramentas AOP é intrinsecamente intrusiva, pois altera o comportamento do código-base de aplicações. A costura de código realizada por compiladores da linguagem AspectJ deve introduzir comportamentos transversais definidos por adendos (*advices*) em programas Java sem causar impactos no seu desempenho.

Neste trabalho caracterizam-se as técnicas de compilação e costura de código adotadas pelos compiladores da linguagem AspectJ *ajc* e *abc*, e identificam-se problemas existentes no código gerado durante a costura de adendos de contorno. Os problemas analisados são a *repetição de implementações de adendos e hachuras (*shadows*)* e *repetição de variáveis de contexto*. Apresentam-se otimizações que eliminam esses problemas para ambos os compiladores, bem como indicações de integração dessas soluções à etapa de costura de código de cada um. As otimizações propostas reduzem o tamanho, o tempo de execução e o consumo de memória de programas AspectJ que usam adendos de contorno.

Agradecimentos

Agradeço aos meus pais, Elias e Vera, a dedicação e oportunidades oferecidas; à Tays as horas de atenção dividida; aos colegas do laboratório LLP, o ambiente de trabalho descontraído e produtivo; aos professores Roberto e Mariza Bigonha, e ao Fabio Tirelo, os inúmeros conselhos e revisões sobre este texto.

Sumário

Capítulo 1

Introdução

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 os requisitos principais de grandes sistemas possam ser implementados de forma modular por meio de construções existentes em linguagens orientadas por objetos, tais como métodos, classes e herança, a introdução de requisitos secundários, embora vitais, nesses sistemas, 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 vida desses sistemas. Em AOP é possível definir esses requisitos transversais de forma modular.

Essa tecnologia 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. Por exemplo, 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.

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 em 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*). Entretanto, como AOP e AspectJ são tecnologias recentes, e estratégias de compilação para progra-

mas escritos em AspectJ não são ainda tão consolidadas quanto técnicas tradicionais de geração de código, tais como análises de variáveis vivas e otimizações *peep-hole*. 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.

Costura de código (*weaving*) é a fase em que se introduz o comportamento de requisitos transversais no programa hospedeiro de uma linguagem orientada por aspectos implementada por ambientes de AOP. Em AspectJ realiza-se apenas costura de código estática, mas outros sistemas permitem ainda costura dinâmica[1]. Costura estática é realizada como um passo de compilação ou durante o carregamento, e modifica o código 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 executável original. A principal diferença entre a costura dinâmica e a estática é que a primeira permite a introdução e remoção de aspectos em tempo de execução, e a última não permite modificar o conjunto de aspectos durante a execução.

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 estática oferece mais possibilidades de otimização, pois permite 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 implementadas por compiladores da linguagem AspectJ, e, a partir dessa caracterização, identificar problemas existentes nessas estratégias e propor soluções para esses problemas.

1.1 Abordagens de Implementação de AOP

Uma implementação de AOP consiste em uma linguagem, que determina construções a serem usadas para descrever requisitos modulares e transversais de programas, e em uma implementação da linguagem, que é responsável por executar programas que usem essa implementação [Lad03, GL03]. Implementações de AOP devem especificar componentes da linguagem usada, como pontos de junção e adendos, e o ambiente de execução da linguagem.

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

O combinador de aspectos (*aspect weaver*) é um componente vital em ambientes de execução de AOP. Um combinador é responsável pela introdução da descrição de requisitos transversais, realizada separadamente em aspectos, no código de um programa. Combinadores normalmente inserem ganchos em pontos da execução do programa, que, quando alcançados, são interceptados para a execução de adendos [SCT03]. A costura de código pode ser realizada em diferentes momentos do ciclo de vida de um programa. A costura pode ser aplicada: (i) durante a compilação (*compile-time weaving*), (ii) durante a execução (*run-time weaving*), e (iii) durante o carregamento (*load-time weaving*).

Costura durante a compilação, também chamada costura estática, é uma modificação física do código do programa, que introduz aspectos diretamente no código executável em que se aplicam. Essa estratégia pode produzir código mais otimizado, já que a etapa de aplicação de aspectos é separada da execução do programa. Código costurado estaticamente, no entanto, é de difícil modificação para remoção ou substituição da política implementada por aspectos; essas tarefas normalmente demandam recompilação de código em sistemas de costura estática.

Em ambientes e aplicações onde modificação e remoção de políticas implementadas por aspectos são necessárias, é interessante a costura em tempo de execução, também chamada de costura dinâmica. Implementações de AOP que aplicam essa estratégia na costura normalmente monitoram a execução de programas em busca da ocorrência de eventos identificados como pontos de junção pelo programador, para que código de aspectos seja introduzido nesses pontos. Ferramentas comumente usadas na costura em tempo de execução são plataformas de depuração (*debugging*) tais como a Java Platform Debugger Architecture[2] (JPDA).

Existem ainda ambientes que implementam AOP e realizam a costura durante o carregamento de construções do programa. Nesses sistemas, a inclusão ou remoção de requisitos transversais em programas não modifica seu código. Essa estratégia possibilita alterar o comportamento de programas sem recompilá-los. Duas implementações de AOP que possuem Java como linguagem hospedeira, AspectWerkz [Bón04] e AspectJ [KHH+01], permitem costura durante o carregamento de classes por meio de *class loaders* [GJSB05, Cap. 12].

Programas implementados em sistemas AOP de costura em tempo de execução, tais como PROSE [PGA02, PAG03] e Wool [SCT03], normalmente possuem desempenho pior do que os que realizam a costura estática sobre o código executável, como AspectJ. O impacto em desempenho causado pela necessidade de monitorar a execução de programas para identificação de pontos de junção pode ser grande, já que os pontos

[2] http://java.sun.com/products/jpda/

de junção do programa podem mudar durante a execução [SCT03]; essa característica contrasta com a costura estática em tempo de compilação, na qual pontos de junção são determinados antes da execução do programa [HH04].

A linguagem AspectJ tradicionalmente suporta costura estática e, em versões mais recentes, o ambiente de execução da linguagem permite também implementar e compilar aspectos separadamente de programas e aplicá-los somente durante o carregamento de classes [Asp05a, Asp05b]. Programas implementados dessa forma são combinados na execução da máquina virtual, por meio de um carregador de classes (*class loader*) personalizado, distribuído como parte do ambiente da linguagem AspectJ. É importante observar que o ambiente Java permanece inalterado apesar da utilização de um carregador de classes personalizado: a especificação da Máquina Virtual Java (JVM) prevê a determinação de diferentes carregadores, especificados pelo usuário, para a execução de programas [LY99, Cap. 5].

1.1.1 AspectJ

Programação Orientada por Aspectos permite a implementação de programas complexos de forma modular, fornecendo maneiras de desenvolver independentemente diferentes requisitos como aspectos desses programas. Aspectos são costurados no código base para prover um único programa com a funcionalidade desejada e 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 a implementação entrelaçada de requisitos, porém apenas no código executável gerado, embora o programador tenha acesso a código modular e de mais 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 costurar 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 necessárias para a compreensão de construções de AspectJ são descritas neste trabalho e outras detalhadas em [Lad03, GL03]. Na terminologia comum de AspectJ[3], 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

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

de uma chamada de método. Um adendo (*advice*), ou comportamento 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*).

Como AspectJ é uma extensão de Java, seu mecanismo de introdução de requisitos transversais é baseado em um compilador que gera código *bytecode* 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 na JVM padrão. 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. Impactos causados pelo código necessário à implementação da costura de código são discutidos na Seção 2.2.1.

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, em vez 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 desobrigar que o código base existente seja recompilado para inclusão de aspectos; na verdade, não é necessário um compilador adicional.

1.1.2 Sistemas com Costura em Tempo de Execução

Costura em tempo de execução é a inclusão de comportamentos transversais implementados por aspectos durante a execução de programas. Nesses sistemas de AOP, aspectos podem ser incluídos, removidos e modificados em um programa durante sua execução, sem recompilação e com pouco ou nenhum recarregamento de código, o que simplifica a implementação de comportamentos transversais cujos requisitos possam mudar durante a execução do programa. Essa costura dinâmica permite ainda realizar testes com aspectos em ambientes em execução antes de associá-los ao sistema em tempo

de compilação. Esta seção apresenta algumas implementações de costura em tempo de execução existentes, especificamente AOP/ST [Böl99], PROSE [PGA02, PAG03], Wool [SCT03] e AspectWerkz [Bón04, Vas04].

AOP/ST

O sistema AOP/ST [Böl99] implementa programação orientada por aspectos para a linguagem Smalltalk. Seu combinador usa herança de classes para costurar código de aspectos em classes de um programa. Cada aspecto que se aplica a uma classe A no programa é transformado em uma subclasse de A, e os métodos afetados por essa aplicação são redefinidos pelo aspecto, encapsulando a implementação definida em A, para implementar o comportamento definido pelo aspecto. Böllert ressalta que a memória necessária para classes geradas durante a costura e o tempo necessário para determinar o método a ser executado, por causa das várias redefinições geradas, são desvantagens no desempenho de AOP/ST [Böl99].

PROSE

PROSE [PGA02, PAG03] é um sistema de AOP dinâmico baseado na introdução de aspectos durante a execução de programas por meio de um compilador Just-In-Time (JIT). Comportamentos ortogonais são costurados em código nativo produzido para os métodos de um programa. Para isso, o programa é compilado inicialmente com ganchos em pontos capturáveis de execução, como chamadas a métodos e acesso a campos de classes. Quando um aspecto é introduzido nesse programa, um gerente de pontos de junção identifica ganchos afetados por esse aspecto, causando a recompilação de código nativo para esses ganchos com chamadas ao novo aspecto. Essa abordagem possui duas desvantagens: um programa sem aspectos possui um *overhead* causado pela execução de ganchos vazios para introdução de aspectos, e a introdução de aspectos sempre causa a recompilação de trechos do código *bytecode* para código nativo.

Programadores descrevem aspectos em PROSE usando uma biblioteca Java em que adendos e tipos de pontos de junção podem ser definidos e combinados. A biblioteca de pontos de junção definida por esse sistema permite criar novas construções AOP e inseri-las dinamicamente em programas em execução.

Wool

O combinador de aspectos do sistema Wool [SCT03] permite ao programador escolher, para cada ponto de junção, se adendos serão executados por meio da plataforma de depuração, ou inseridos como ganchos no programa via modificação e recarregamento de código. Inicialmente, Wool inclui ganchos como pontos de parada (*breakpoints*) nos

pontos de junção do programa, e, quando esses pontos são atingidos, a execução da máquina virtual é interrompida para a execução de adendos via JPDA (Java Platform Debug Architecture). Entretanto, a interrupção da máquina virtual é uma operação cara em termos de tempo de execução, e pode causar impacto negativo no desempenho do programa, especialmente se pontos do programa para os quais nenhum adendo é ativado forem alcançados freqüentemente. Para que pontos de junção executados freqüentemente tenham um desempenho melhor, Wool permite costurar o código executável de métodos para inclusão de código de adendos; o código costurado substitui o original e remove o ponto de parada de depuração original. A abordagem de gerar código nativo apenas para trechos de código afetados por aspectos e executados freqüentemente evita o impacto de várias compilações quando muitos aspectos são inseridos dinamicamente em programas.

Wool é implementado como uma biblioteca Java que define formas de implementar aspectos e um combinador usado durante a execução. Na definição de um aspecto usando Wool, o programador possui controle sobre a costura de código e sobre o método de costura, via pontos de parada ou chamadas de método embutidas, a ser utilizada.

AspectWerkz

O arcabouço AspectWerkz [Bón04, Vas04] implementa uma "costura centrada em pontos de junção" que permite incluir e remover aspectos de programas em execução costurados *estaticamente*, dado que os pontos de junção não mudem. Para isso, a costura inclui um nível de indireção em cada método que é um ponto de junção, chamando um "gerente de pontos de junção" responsável por ativar adendos aplicáveis a esse ponto em tempo de execução. Nessa abordagem não há informações sobre adendos aplicados no código costurado, de forma que aspectos podem ser incluídos e removidos por meio do gerenciador de pontos de junção durante a execução do programa.

Uma costura em duas fases é implementada para permitir alterações no conjunto de pontos de junção do programa durante sua execução. Na primeira fase, chamada de fase de preparação de classes, um arquivo de configuração é lido para determinar quais classes devem ser preparadas. Na fase de ativação, o código *bytecode* é transformado para aceitar a aplicação de pontos de junção, definidos durante a execução, e executar os aspectos correspondentes. A costura em tempo de execução é compatível com a costura estática realizada durante o carregamento pelo AspectWerkz [Vas04].

1.1.3 Escopo Deste Estudo

Costura em tempo de execução é a inclusão de comportamentos transversais implementados por aspectos durante a execução de programas. Nesses sistemas de AOP, aspectos

podem ser incluídos, removidos e modificados em um programa durante sua execução, sem recompilação e com pouco ou nenhum recarregamento de código, o que simplifica a implementação de comportamentos transversais cujos requisitos possam mudar durante a execução do programa. Essa costura dinâmica permite ainda realizar testes com aspectos em ambientes em execução antes de associá-los estaticamente, ou seja, durante a compilação, ao sistema. Os sistemas AOP/ST [Böl99], PROSE [PGA02, PAG03], Wool [SCT03] e AspectWerkz [Bón04, Vas04] são exemplos de costura em tempo de execução.

A linguagem AspectJ 5 [Asp05b] não suporta costura em tempo de execução. Aspectos definidos em AspectJ podem ser costurados durante a compilação, após a compilação e até durante o carregamento de classes, porém o conjunto de aspectos e pontos de junção do programa não pode ser modificado durante sua execução [Asp05a]. Implementações de AOP que realizam costura estática, seja ela durante a compilação ou durante o carregamento, normalmente geram programas mais eficientes do que aquelas em que a costura é realizada durante a execução [PAG03].

Costura dinâmica apresenta a vantagem de permitir que requisitos transversais ou suas implementações (aspectos) mudem durante a execução de um programa. Essa característica é necessária para implementar aspectos que definem políticas que podem mudar durante a execução do programa, tais como, por exemplo, políticas de armazenamento em *cache* e de monitoramento de programas [SCT03].

Na costura dinâmica, otimizações podem ser restringidas quanto ao escopo, já que as análises necessárias para realizar otimizações em todo o programa podem tomar tempo considerável da sua execução, sendo necessário determinar um equilíbrio entre a vantagem obtida por determinada otimização e o custo necessário para realizá-la durante a execução.

Neste trabalho optou-se por priorizar o estudo de técnicas de costura estática, realizada como uma etapa integrada à compilação ou durante o carregamento de código, já que sistemas que implementam essas técnicas oferecem maior margem à otimização: como costura estática é um processo independente da execução do programa, análises complexas executadas nessa fase podem produzir ganhos significativos em relação ao tamanho do código gerado e ao seu desempenho.

A linguagem AspectJ foi escolhida como objeto de estudo deste trabalho por ser a ferramenta AOP mais usada na indústria, e por ser implementável, com algumas modificações, por técnicas tradicionais de compilação. Os problemas estudados são aplicados na costura estática realizada pelos compiladores dessa linguagem.

1.2 Objetivos e Problemas Abordados

O objetivo deste trabalho é, a partir de um levantamento das técnicas atuais de costura de código implementadas por compiladores da linguagem AspectJ, identificar e solucionar problemas encontrados nessas técnicas. Para isso descrevem-se a arquitetura e as técnicas de costura de código adotada por esses compiladores, e analisa-se código gerado por eles.

Descrevem-se nesta dissertação otimizações aplicáveis ao código gerado por compiladores da linguagem AspectJ a partir de programas que usam adendos de contorno. Os problemas tratados são, especificamente, a repetição de implementações de adendos e hachuras (veja Capítulo 2), e repetição de variáveis de contexto.

O problema de repetição de adendos e hachuras surge quando uma classe c possui n hachuras de um adendo de contorno a, situação em que n pares $(adendo, hachura)$ de métodos são introduzidos no código *bytecode* produzido para essa classe. A solução para esse problema consiste em eliminar os $n-1$ pares desnecessários do código gerado para a classe c e substituir referências a eles por referências aos seus correspondentes remanescentes, reduzindo assim o tamanho do código gerado para programas AspectJ que usem adendos de contorno.

O problema de variáveis de contexto repetidas também aparecem em código gerado durante a costura de adendos de contorno. São resultantes da captura separada de variáveis locais necessárias para a correta execução de hachuras e das variáveis explicitamente capturadas pelo programador em pontos de junção por meio das cláusulas **args**, **target** ou **this**. Esse problema causa aumentos desnecessários no tamanho, no tempo de execução e no consumo de memória de programas AspectJ.

Ambas as otimizações propostas promovem redução no tamanho do código gerado para programas AspectJ e, além disso, a eliminação de variáveis de contexto repetidas promove maior eficiência em tempo de execução e consumo de memória em programas que usam adendos de contorno.

As contribuições deste trabalho são:

- identificação e descrição das técnicas de costura de código adotadas pelos compiladores *ajc* e *abc*;
- identificação de problemas existentes nessas técnicas;
- propostas de soluções para os problemas de implementações repetidas de adendos e variáveis de contexto repetidas, promovendo redução no tamanho, tempo de exeução e consumo de memória do código *bytecode* gerado para programas AspectJ;

- integração dessas soluções aos compiladores estudados.

1.3 Organização Deste Texto

Capítulo 2 apresenta a linguagem AspectJ, seus compiladores e as técnicas de costura de código adotadas por eles. Problemas identificados nessas técnicas são descritos no Capítulo 3, e medologias para solução desses problemas são apresentados separadamente nos Capítulos 4 e 5. Os resultados obtidos são resumidos no Capítulo 6.

Capítulo 2

Implementações de AspectJ

A primeira versão da linguagem AspectJ foi lançada em 2001, como um projeto do laboratório PARC/Xerox. A linguagem está em constante desenvolvimento, embora suas construções e conceitos básicos tenham se solidificado. Este capítulo apresenta uma breve introdução à linguagem AspectJ e aos compiladores *ajc* e *abc*.

2.1 A Linguagem AspectJ

Esta seção tem o objetivo de apresentar a linguagem AspectJ e suas construções por meio de exemplos pequenos e objetivos. Fogem do escopo deste texto descrições detalhadas da sintaxe e da semântica da linguagem, que podem ser encontradas em [Aspb, KHH+01, Lad03, GL03].

AspectJ é uma extensão da linguagem Java que permite a inclusão de comportamento dinâmico em pontos definidos da execução de programas, bem como inclusão estática de elementos, como métodos e atributos, em classes existentes. O modelo de pontos de junção da linguagem AspectJ é *léxico*, que identifica entidades e configurações de execução de programas Java por seus nomes. Entidades, como classes e métodos, são artefatos estáticos de um programa. Configurações são artefatos dinâmicos da execução desse programa, como o fluxo de chamadas de métodos.

Requisitos transversais são implementados em AspectJ em estruturas similares a classes, e permitem interceptar e modificar pontos da execução de programas. As seções a seguir apresentam separadamente as construções da linguagem AspectJ.

2.1.1 Pontos de Junção e Conjuntos de Junção

Pontos da execução do programa, chamados de pontos de junção (*join points*), são pontos identificáveis da sua execução. Listagem 2.1 mostra um programa Java simples, e seus pontos de junção são listados na Tabela 2.1. Note que algumas linhas do

programa contêm mais de um ponto de junção como pontos de junção do programa, mas esse tipo de ponto de junção não aparece neste exemplo.

```
1  public abstract class Figure {
2      protected int x,y;
3      public Figure(int x, int y) { setX(x); setY(y); }
4      public int getX() { return x; }
5      public int getY() { return y; }
6      public void setX(int x) { this.x = x; }
7      public void setY(int y) { this.y = y; }
8      public String toString() {
9          return "(" + x + "," + y + ")";
10     }
11 }
12
13 public class Circle extends Figure {
14     private int radius;
15     public Circle(int x, int y, int radius) throws Exception {
16         super(x,y);
17         setRadius(radius);
18     }
19     public int getRadius() { return radius; }
20     public void setRadius(int radius) throws Exception {
21         if (radius < 0)
22             throw new Exception();
23         this.radius = radius;
24     }
25 }
26
27 public class JoinPointShowcase {
28     static java.io.PrintStream out;
29     static {
30         out = System.out;
31     }
32     public static void main(String[] args) {
33         try {
34             Circle c = new Circle(0,20,5);
35             c.setX(30);
36         } catch(Exception e) {
37             out.println("Illegal coordinate or radius value");
38         }
39     }
40 }
```

Listagem 2.1: Exemplo de um programa com vários pontos de junção em Java.

Tipo de ponto de junção	Ocorrências
Acesso a um atributo para escrita (*field-set*)	Linhas 6, 7, 23 e 30
Acesso a um atributo para leitura (*field-get*)	Linhas 4, 5, 9, 19, 30 e 37
Chamada a um método (*method-call*)	Linhas 3, 17, 35, e 37
Execução de um método (*method-execution*)	Corpo dos métodos das classes Figure, Circle e JoinPointShowcase
Chamada a um construtor	Linhas 22 e 34
Execução de um construtor	Corpos dos construtores de Figure e Circle
Inicialização estática (*static-initialization)*	Bloco estático da classe JoinPointShowcase, nas linhas 29 a 31
Inicialização de objetos (*initialization*)	Blocos não-estáticos externos a métodos de todas as classes e corpos de construtores
Pré-inicialização de objetos (*pre-initialization*)	Construção de argumentos para chamada **super** do construtor de Circle
Tratamento de exceção (*handler*)	Bloco **catch** do método main

Tabela 2.1: Pontos de junção.

Pontos de junção são interceptados, ou capturados, em AspectJ por meio de expressões de conjuntos de junção (*pointcuts*). Conjuntos de junção podem ser declarados em aspectos e classes, e podem ser abstratos. A expressão a seguir captura chamadas ao método setX da classe Figure:

```
call(void Figure.setX(int))
```

Expressões "curinga" (*wildcards*) também podem ser usadas para declarar conjuntos de junção genéricos; os seguintes curingas são aceitos:

- $*$: qualquer número de caracteres válidos no nome de uma classe;
- $..$: qualquer número de pacotes;
- $+$: qualquer subclasse ou subinterface de classes e interfaces.

A seguinte expressão captura chamadas a métodos de Figure e suas subclasses, cujos nomes começam com *"set"*, e com quaisquer parâmetros:

```
call(void Figure+.set*(..))
```

Chamadas a construtores também são interceptadas por meio da cláusula call. Entretanto, em vez de o nome do método e seu tipo de retorno, padrões para chamadas de construtores contêm o nome da classe seguido da palavra reservada **new**, como mostra a expressão a seguir, que intercepta chamadas ao construtor de Figure:

```
call(Figure.new(int,int))
```

A costura de conjuntos de junção pode ser restringida por meio das cláusulas **within** e **withincode**. A cláusula **within** restringe a aplicação de conjuntos de junção a uma classe, e **withincode** a restringe a métodos. As operações de conjunção (&&), disjunção(||) e negação (!) são usadas para combinar conjuntos de junção. A expressão a seguir intercepta acessos de escrita ao atributo x da classe Figure que ocorram no corpo dessa classe, mas fora do método setX:

```
set(int Figure.x) && within(Figure) && !withincode(void Figure.setX(int))
```

É possível ainda interceptar pontos de junção com base na pilha de execução de programas. A cláusula **cflow** captura pontos que ocorram enquanto houver uma ativação viva de outro determinado ponto na pilha de execução do programa. A expressão a seguir intercepta acessos de escrita à variável x da classe Figure que ocorram no fluxo de execução de um método moveBy:

```
set(int Figure.x) && cflow(execution (void Figure+.moveBy(int dx, int dy)))
```

Uma cláusula similar a essa, **cflowbelow**, captura pontos de junção que ocorram no fluxo de execução de um determinado ponto, mas em um nível diferente da pilha. Ou seja, a cláusula **cflowbelow** não é aplicada quando os dois pontos de junção associados a ela ocorrerem na mesma ativação de método. No exemplo a seguir, chamadas ao método setX da classe Figure e suas subclasses são ignoradas quando ocorrerem no corpo do próprio método main.

```
call(void Figure+.setX(int)) &&
    cflowbelow(execution (static void main(String[])))
```

Em conjuntos de junção pode-se ainda usar as cláusulas **execution**, que intercepta todo o corpo de um método ou construtor, **staticinitialization** , para o bloco de inicialização estática de uma classe, **initialization** , para a inicialização de um objeto, e **preinitialization** , para comandos executados na construção da chamada ao construtor da superclasse (carga de parâmetros).

Conjuntos de junção podem ser declarados e associados a um nome; o exemplo a seguir mostra a declaração do conjunto de junção setXCalls:

```
pointcut setXCalls(): call(void Figure.setX(int));
```

O par de parênteses que aparece próximo ao nome de um conjunto de junção em sua declaração é usado para *captura de contexto*. As cláusulas **args**, **this** e **target** capturam, respectivamente, os argumentos, o objeto em execução e o alvo de uma chamada. A semântica exata de cada uma dessas cláusulas depende do tipo de conjunto de junção a que a captura de contexto está associada: a cláusula **args**, por exemplo, captura argumentos de chamadas a métodos, quando associada a conjuntos de junção do tipo **call** , e o valor atribuído a um campo quando associada a **set**. O exemplo a seguir

modifica o conjunto de junção setXCalls para capturar os argumentos de chamadas ao método setX:

```
pointcut setXCalls(int v): call(void Figure.setX(int)) && args(v);
```

Note que variáveis capturadas devem estar "ligadas" ao conjunto de junção na sua declaração para que possam ser usadas nas cláusulas de captura, como no exemplo anterior. Pode-se ainda associar conjuntos de junção, como no exemplo a seguir, que captura chamadas a setX que estejam fora da classe Figure, usando o conjunto de junção já declarado setXCalls:

```
pointcut setXCallsOutsideFigure(int v): setXCalls(v) && !within(Figure)
```

Conjuntos de junção apenas determinam os ganchos de programas em que as implementações de requisitos transversais serão ativados. Esses requisitos são implementados, em AspectJ, por meio de adendos e aspectos. Seção 2.1.2 apresenta essas construções.

2.1.2 Aspectos e Adendos

Aspectos são entidades de alto nível para implementação de comportamento transversal em AspectJ. Eles são similares a classes, pois podem declarar métodos e atributos, além de conjuntos de junção. Aspectos podem também inserir novos métodos e atributos em classes, e modificar sua hierarquia, acrescentando a sua assinatura novas interfaces e uma superclasse, respeitando as restrições de herança da linguagem Java – essa característica, chamada transversalidade estática, é descrita na Seção 2.1.3.

Listagem 2.2 apresenta um aspecto que captura chamadas ao método setX da classe Figure, e mostra o valor de seus argumentos. Nesse aspecto, declaram-se um conjunto de junção e um adendo. Adendos (*advices*) são trechos de código associados a conjuntos de junção. Quando a execução de um programa passa por um determinado ponto de junção, adendos associados a esse ponto por meio de conjuntos de junção são ativados. Figura 2.1 é uma representação gráfica do processo de ativação do adendo mostrado na Listagem 2.2. Nessa figura, a seta contínua representa o fluxo normal de execução do programa, e setas tracejadas mostram a ativação de um adendo. A chamada a setX do programa principal é interceptada pelo conjunto de junção setXCalls, o que causa a ativação do adendo associado a ele.

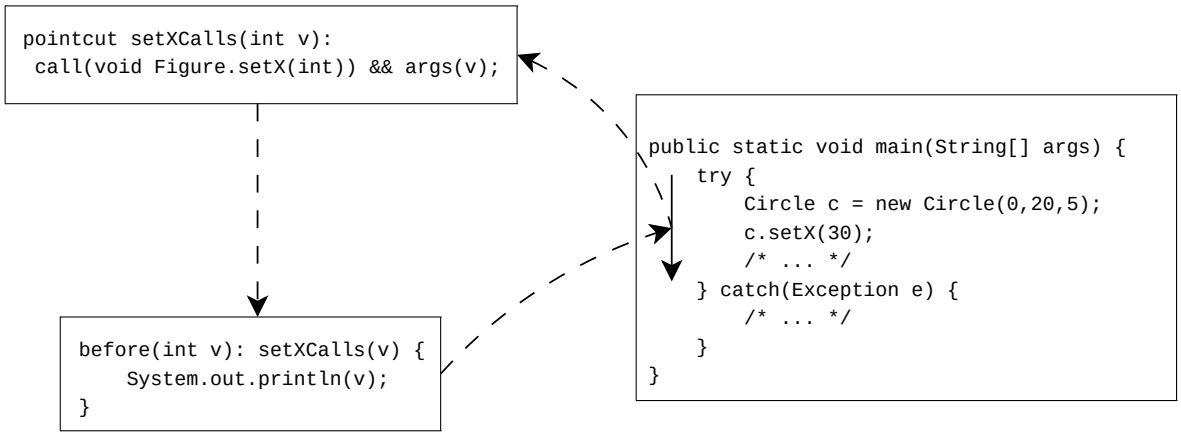

Figura 2.1: Representação gráfica da ativação de um adendo.

Note, no código da Listagem 2.2, que o tipo de um adendo é determinado na sua declaração; nesse exemplo, a marcação é a palavra-chave **before**. Adendos em AspectJ podem ser anteriores (*before*), posteriores (*after*) ou de contorno (*around*). Na declaração do adendo, associa-se um conjunto de junção que indica os pontos da execução de um programa em que o adendo deve atuar. Adendos anteriores, como o da Listagem 2.2, são executados antes do ponto de junção associado a eles, e adendos posteriores são executados depois desse ponto. Adendos de contorno substituem completamente a execução de pontos de junção associados a eles.

```
1 public aspect XCallsAspect {
2     pointcut setXCalls(int v): call(void Figure.setX(int)) && args(v);
3     before(int v): setXCalls(v) {
4         System.out.println(v);
5     }
6 }
```

Listagem 2.2: Aspecto simples.

Ainda no adendo declarado na Listagem 2.2, note que variáveis de contexto capturadas por conjuntos de junção devem também ser declaradas no adendo. Adendos podem definir conjuntos de junção anônimos, criados em sua declaração. O trecho de código a seguir mostra a declaração de um adendo que captura variáveis de contexto e usa um conjunto de junção anônimo.

```
before(int v, Figure f):
    call(void Figure.setX(int)) && args(v) && target(f) {
```

```
    /* ... */
}
```

Adendos anteriores são os mais simples de AspectJ, por serem sempre executados antes de seus pontos de junção. Adendos posteriores são executados após seus pontos de junção, mas sua aplicação pode ser restringida a retornos com ou sem lançamento de exceção. Pode-se ainda capturar o valor retornado ou a exceção lançada pela execução do ponto de junção. Os seguintes adendos posteriores são exemplos de restrição da aplicação quanto ao resultado de pontos de junção. No primeiro, capturam-se objetos resultantes de chamadas ao construtor de subclasses de Figure, e no segundo capturam-se exceções ocorridas durante a execução do método setRadius da classe Circle.

```
after() returning(Figure f): call(Figure+.new(..)) {
    System.out.println(f);
}
after(int v) throwing(Exception e):
    call(void Circle.setRadius(int)) && args(v) {
    System.err.println("Exception occurred with argument " + v);
    e.printStackTrace();
}
```

É possível também criar adendos posteriores sem restringir sua aplicação quanto ao retorno, como no exemplo a seguir, que é executado sempre no fim do método main, independentemente da ocorrência de exceções:

```
after(): execution(static void JoinPointShowcase.main(String[])) {
    System.out.println("Bye!");
}
```

Os adendos de contorno são os mais poderosos da linguagem AspectJ. Com esse tipo de adendo, é possível obter os mesmos comportamentos dos adendos anteriores e posteriores. Adendos de contorno substituem totalmente a execução de seus pontos de junção, mas podem redirecionar a execução do programa para esses pontos por meio do comando **proceed**. Como adendos de contorno são executados em vez de seus pontos de junção, eles devem retornar um valor do mesmo tipo de pontos que interceptam. O exemplo a seguir mostra um adendo de contorno que impede a modificação do raio de um círculo:

```
void around(): set(int Circle.radius) { }
```

Variáveis de contexto capturadas por adendos de contorno devem ser passadas, por meio do comando **proceed**, aos seus pontos de junção. O adendo pode também usar o comando **proceed** para executar o ponto de junção interceptado com outras variáveis de contexto, modificando sua semântica. O exemplo a seguir mostra um adendo que impede a atribuição de valores negativos ao raio de círculos:

```
void around(int v): call(void Circle.setRadius(int)) && args(v) {
    if (v >= 0)
        proceed(v);
}
```

O comando **proceed** não precisa aparecer diretamente no corpo de um adendo de contorno; ele pode aparecer também em uma classe anônima criada no corpo do adendo. Essa característica confere grande flexibilidade à aplicação de adendos de contorno. Considere o aspecto da Listagem 2.3, que captura atribuições ao campo radius da classe Circle.

Com esse aspecto, modificações do valor de radius são transformadas em comandos, e armazenadas em um registro de comandos (CommandRegistry) para implementar transparentemente as funcionalidades desfazer e refazer, que navegam no histórico de atualizações do valor dessa variável. A interface Command define métodos undo e redo, para desfazer e refazer um comando. O registro de comandos usa duas pilhas de comandos: a primeira, undoable, armazena os comandos que podem ser desfeitos, e redoable armazena os que podem ser refeitos.

No adendo de contorno definido em UndoRedoAspect, cria-se uma implementação anônima de um comando, que contém modificações ao valor de radius no corpo dos métodos undo e redo. Em seguida executa-se o comando **proceed** uma vez, para que a modificação no valor de radius seja efetivada. O comando criado é adicionado ao registro, para que possa ser desfeito ou refeito. Pode-se então chamar os métodos undo e redo da classe CommandRegistry para desfazer e refazer modificações a essa variável.

Esse exemplo mostra um fenômeno comum em programas AspectJ: aspectos, mesmo para implementação de comportamentos tranversais complexos, são freqüentemente simples e de poucas linhas. A complexidade intrínseca à implementação desses requisitos em orientação por objetos é absorvida pelo combinador de aspectos da linguagem.

As construções descritas nesta seção são usadas para modificar o comportamento dinâmico de programas usando AspectJ. Seção 2.1.3 descreve os mescanismos dessa

linguagem para se modificar a estrutura de programas e declarar condições de erro que impeçam sua compilação.

2.1.3 Transversalidade Estática e Outras Construções

As construções apresentadas nas Seções 2.1.1 e 2.1.2 permitem modificar o comportamento dinâmico de programas. Transversalidade estática (*static crosscutting*) é o mecanismo da linguagem AspectJ para modificação da estrutura de programas.

Há quatro tipos de transversalidade estática: inserção estática, modificação da hierarquia de classes, introdução de erros e advertências, e suavização de exceções. Inserção estática é a inserção de atributos e métodos em classes, interfaces e aspectos existentes. O programador pode determinar situações em que erros e advertências devem ser lançados pelo compilador, com o objetivo de garantir políticas de desenvolvimento. É possível ainda transformar exceções verificadas (subclasses de Exception) em exceções não-verificadas (subclasses de RuntimeException), para se implementar uma estratégia de tratamento de exceções que não polua o código de métodos do sistema.

Embora essas construções sejam de grande importância em programas escritos na linguagem AspectJ, elas estão fora do escopo das otimizações propostas neste trabalho. Optou-se, assim, por não apresentar as construções relacionadas classificadas como transversalidade estática nesta introdução. Essas construções, e usos comuns para elas, são apresentadas em profundidade em [Lad03, GL03].

2.1.4 Notação @AspectJ

A notação apresentada neste capítulo é a notação tradicional da linguagem AspectJ, também chamada *code-style AspectJ*. Com o advento de anotações, introduzidas na Versão 5.0 da linguagem Java, os desenvolvedores das linguagens AspectJ possibilitaram a implementação de aspectos e adendos usando apenas as construções da linguagem Java. Dessa forma, o uso de AOP na linguagem Java deixa de depender de novas construções. Essa nova notação é normalmente chamada de @AspectJ, ou ainda @AJ, em uma alusão ao marcador '@' usado para criar anotações em Java 5.0.

Escolher entre as notações *code-style* e @AspectJ não causa impacto na implementação de requisitos transversais, já que ambas possuem o mesmo poder [Asp05b]. Neste trabalho, usa-se a notação *code-style* por ser mais sucinto e legível: o exemplo da Listagem 2.5, extraído de [Asp05b], mostra um mesmo conjunto de junção definido em ambas as notações.

```
1  public aspect UndoRedoAspect {
2      pointcut radiusSetCommands(int v, Circle c):
3          set(int Circle.radius) && args(v) && target(c);
4      void around(final int v, final Circle c): radiusSetCommands(v, c) {
5          final int oldValue = c.getRadius();
6          CommandRegistry.addCommand(
7                  new CommandRegistry.Command() {
8                      public void undo() { proceed(oldValue,c); }
9                      public void redo() { proceed(v,c); }
10                 }
11         );
12         proceed(v,c);
13     }
14 }
```

Listagem 2.3: Aspecto para criação de comandos.

```
1  public class CommandRegistry {
2      private static Stack undoable = new Stack();
3      private static Stack redoable = new Stack();
4      public static void addCommand(Command c) {
5          redoable.removeAllElements();
6          undoable.push(c);
7      }
8      public static void undo() {
9          if (!undoable.empty()) {
10             Command c = (Command) undoable.pop();
11             c.undo();
12             redoable.push(c);
13         }
14     }
15     public static void redo() {
16         if (!redoable.empty()) {
17             Command c = (Command) redoable.pop();
18             c.redo();
19             undoable.push(c);
20         }
21     }
22     public static abstract class Command {
23         public abstract void undo();
24         public abstract void redo();
25     }
26 }
```

Listagem 2.4: Registro de comandos.

```
1 // Notação tradicional
2 pointcut someCallWithIfTest(int i):
3     call(* *.*(int)) && args(i) && if(i > 0);
4
5 // Notação @AspectJ
6 @Pointcut("call(* *.*(int)) && args(i) && if()")
7 public static boolean someCallWithIfTest(int i) {
8     return i > 0;
9 }
```

Listagem 2.5: Conjunto de junção definido em ambas as notações permitidas em AspectJ.

2.2 Compilação de Programas em AspectJ

AspectJ é uma extensão da linguagem Java. Os compiladores *ajc* [Aspb] e *abc* [ABCdg] aceitam código Java e AspectJ como entrada, e produzem código *bytecode* na saída, que pode ser executado na Máquina Virtual Java padrão. Para isso, todo programa AspectJ deve ser acompanhado pela biblioteca *runtime* do compilador que o gerou, de maneira que as dependências introduzidas pelos compiladores durante a costura de código sejam resolvidas.

A compilação de programas em linguagens orientadas por aspectos é um tema recente, e há poucos compiladores conhecidos para AspectJ. O compilador *ajc*, oficial da linguagem, em sua primeira versão, manipulava o código-fonte Java de programas para introduzir comportamento de aspectos, mas logo passou a realizar a costura de código diretamente em *bytecode*. O compilador *abc* também manipula *bytecode* para introduzir código de aspectos em classes.

Compiladores AspectJ são independentes de um compilador Java, podendo compilar programas escritos puramente em Java. Quando aspectos são compilados junto a código Java, uma etapa adicional é realizada durante a compilação, chamada *costura de código*. Nessa etapa, realizada pelo que se chama um combinador (*weaver*), as especificações de comportamentos transversais definidas em aspectos são costuradas, ou combinadas, ao código *bytecode* gerado para as classes do programa.

Simplificando o processo de compilação, pode-se dizer que em ambos os compiladores o *front-end* lê código-fonte de classes e aspectos e gera uma AST e um conjunto de informações para a costura de código. O *back-end* então usa a AST do programa para gerar código *bytecode*. Em seguida, o combinador usa as informações de costura produzidas pelo *front-end* para modificar o *bytecode* gerado pelo *front-end*, introduzindo o comportamento determinado pelos aspectos do programa em suas classes.

Técnicas tradicionais de compilação são aplicadas nas etapas de análise léxica, sintática e semântica e na geração e otimização de código de compiladores da linguagem AspectJ. Esta seção apresenta em detalhes as técnicas de costura de adendos, por serem próprias da compilação de linguagens orientadas por aspectos. Omite-se deste capítulo a descrição das técnicas de implementação de construções que modificam estaticamente a estrutura de programas, como inserção estática e suavização de exceções. A compilação dessas construções é simples, quando comparada à costura de adendos, e está fora do escopo das otimizações propostas neste trabalho.

2.2.1 Costura de Adendos

O foco deste trabalho está nas estratégias de costura de compiladores de AspectJ. Este estudo é baseado em código *bytecode* contido em arquivos de classes gerados tanto por *ajc* quanto por *abc*[1]. *Bytecode* é um código de pilha, baseado na existência de um repositório de constantes para referências a classes, métodos e campos. Uma descrição completa da estrutura desse código pode ser encontrada em [LY99]. 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 que geram.

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

Aspectos são transformados em classes Java, e adendos declarados no seu corpo 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 métodos que implementam adendos nos locais apropriados. Código de chamada de adendos pode incluir *resíduo dinâmico*, que consiste de código para testes a serem 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*.

Para simplificar a discussão sobre técnicas de costura de adendos, apresenta-se novamente o exemplo proposto na Seção 2.1. Listagem 2.6 reapresenta o código desse exemplo; o restante desta seção mostrará o efeito das técnicas de costura aplicadas a adendos definidos sobre esse programa. Listagem 2.7 mostra o código *bytecode* gerado

[1] Código *bytecode* é apresentado neste texto em sua representação textual, tal como a construída pelo programa *javap*, disponível com o Java SDK.

para o método main da classe Figs, quando não há aspectos no programa[2]. Exceto quando explicitamente destacado, trechos de código *bytecode* mostrado nas listagens desta seção foram gerados pelo compilador *ajc*.

```
1  public abstract class Figure {
2      protected int x,y;
3      public Figure(int x, int y) { setX(x); setY(y); }
4      public int getX() { return x; }
5      public int getY() { return y; }
6      public void setX(int x) { this.x = x; }
7      public void setY(int y) { this.y = y; }
8      public String toString() {
9          return "(" + x + "," + y + ")";
10     }
11 }
12 public class Circle extends Figure {
13     private int radius;
14     public Circle(int x, int y, int radius) throws Exception {
15         super(x,y);
16         setRadius(radius);
17     }
18     public int getRadius() { return radius; }
19     public void setRadius(int radius) throws Exception {
20         if (radius < 0)
21             throw new Exception();
22         this.radius = radius;
23     }
24 }
25 public class Figs {
26     static java.io.PrintStream out;
27     static {
28         out = System.out;
29     }
30     public static void main(String[] args) {
31         try {
32             Circle c = new Circle(0,20,5);
33             c.setX(30);
34         } catch(Exception e) {
35             out.println("Illegal coordinate or radius value");
36         }
37     }
38 }
```

Listagem 2.6: Exemplo de um programa com vários pontos de junção em Java.

[2] Tipos de métodos e campos são determinados em *bytecode* por *strings* descritores. Por exemplo, na posição 8 do código da Listagem 2.7, o *string* (III)V indica que o método construtor da classe Circle, <init>, recebe três parâmetros inteiros e tem retorno vazio. O formato desses *strings* é detalhado na Seção 4.3.2.1

```
public static void main(java.lang.String[]);
  Code:
   0:   new #26; //class Circle
   3:   dup
   4:   iconst_0 //Primeiro argumento
   5:   bipush  20 //Segundo argumento
   7:   iconst_5 //Terceiro argumento
   8:   invokespecial   #29; //Method Circle."<init>":(III)V
   11:  astore_1
   12:  aload_1 //Carga de objeto Circle
   13:  bipush  30 //Primeiro argumento
   15:  invokevirtual   #33; //Method Circle.setX:(I)V
   18:  goto    34
   21:  astore_1
   22:  getstatic   #14; //Field out:Ljava/io/PrintStream;
   25:  ldc #35; //String "Illegal coordinate or radius value"
   27:  invokevirtual   #41; //Method java/io/PrintStream.println:(Ljava/
      lang/String;)V
   30:  aload_1
   31:  invokevirtual   #46; //Method java/lang/Exception.printStackTrace
      :()V
   34:  return
  Exception table:
   from   to  target type
     0    21    21   Class java/lang/Exception
```

Listagem 2.7: *Bytecode* do método Figs.main.

Suponha que o aspecto TestAspect, definido na Listagem 2.8, seja acrescentado a esse programa. O adendo anterior (*before advice*) definido nesse aspecto é executado antes de chamadas ao método setX da classe Figure e suas subclasses, como da linha 33 da Listagem 2.6. No *bytecode* gerado para o método main, essa hachura corresponde às instruções nas posições 13 e 15[3].

```
1 public aspect TestAspect {
2     before(): call(void Figure.setX(int)) {
3         System.out.println("calling setX");
4     }
5 }
```

Listagem 2.8: Aspecto que define um adendo anterior.

[3] Instruções em código *bytecode* são representadas por *opcodes* de um *byte*, seguidos de qualquer número de argumentos. As posições mostradas nas representações textuais de métodos em *bytecode* correspondem ao índice do *opcode* dessas instruções no corpo do método.

Adendos anteriores são 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; o exemplo da Listagem 2.8 usa a primeira. Em conjuntos de junção de execução, uma chamada ao método de adendo é incluída 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. Listagem 2.9 mostra o código gerado no código de TestAspect para o adendo definido na Listagem 2.8, e Listagem 2.10 mostra o trecho do código do método main modificado para chamar esse adendo.

```
public void ajc$before$TestAspect$1$d147c5c6();
  Code:
   0:   getstatic   #34; //Field java/lang/System.out:Ljava/io/
        PrintStream;
   3:   ldc #36; //String "calling setX"
   5:   invokevirtual   #42; //Method java/io/PrintStream.println:(Ljava/
        lang/String;)V
   8:   return
```

Listagem 2.9: *Bytecode* gerado para o adendo da Listagem 2.8 e sua aplicação em Figs.main.

```
public static void main(java.lang.String[]);
  Code:
   /* ... */
   13:  bipush  30
   15:  invokestatic    #60; //Method TestAspect.aspectOf:()LTestAspect;
   18:  invokevirtual   #63; //Method TestAspect.
        ajc$before$TestAspect$1$d147c5c6:()V
   21:  invokevirtual   #33; //Method Circle.setX:(I)V
   /* ... */
```

Listagem 2.10: *Bytecode* gerado para o adendo da Listagem 2.8 e sua aplicação em Figs.main.

Note, no código costurado do método main, que a instrução que efetivamente chama o método setX, que antes encontrava-se na posição 15, foi deslocada pela inserção de uma chamada ao adendo anterior aplicado a ela para a posição 21.

Se o conjunto de junção associado ao adendo da Listagem 2.8 for modificado para usar a cláusula **execution**, e com isso interceptar execuções do método setX em vez de chamadas a ele, a chamada ao adendo não é mais inserida no corpo de main, mas sim,

no corpo do próprio setX, como mostra a Listagem 2.11. O método setX original contém apenas as instruções das posições 6 a 11 dessa listagem.

Adendos posteriores (*after advices*) possuem a mesma distinção quanto a conjuntos de junção de chamada e de execução. Esses adendos, no entanto, possuem uma peculiaridade: sua aplicação pode ser restringida a execuções de métodos que retornem normalmente (*after returning*) ou que lancem uma exceção (*after throwing*).

O adendo posterior da Listagem 2.12, declarado em TestAspect, é executado após chamadas ao método setX da classe Figure. O *bytecode* gerado para aplicá-lo à chamada da linha 33 é mostrado na Listagem 2.13. Nessa listagem, há duas porções de código idênticas que implementam a aplicação do adendo: as instruções das posições 22 e 25, e posições 31 e 34. O primeiro par de instruções corresponde à aplicação do adendo quando ocorre uma exceção, e o segundo, quando setX retorna normalmente. Se a aplicação desse adendo fosse restringida por meio das palavras-chave **returning** ou **throwing**, uma dessas aplicações seria omitida de acordo com o caso.

Se um adendo posterior se aplica a um conjunto de junção de chamada, como no exemplo da Listagem 2.12, uma chamada a ele é inserida após chamadas ao método selecionado; se ele se aplica a um conjunto de junção de execução, o adendo é chamado no fim 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 fim* do método, de modo que apenas uma chamada ao adendo posterior é inserida.

Adendos de contorno substituem a execução dos pontos de junção a que 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 executa o ponto de junção original interceptado. Como o mesmo adendo pode ser aplicado a vários locais do código, o "código original" pode ser um de muitos pontos de junção do programa. Os compiladores *ajc* e *abc* adotam diferentes estratégias na costura de comportamentos transversais desse tipo, que são descritas nas Seções 2.2.1.1 e 2.2.1.2, respectivamente. A costura de adendos que usam a cláusula *cflow* é descrita na Seção 2.2.1.4.

2.2.1.1 Adendos de Contorno no Compilador *ajc*

A estratégia para costurar adendos 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]. Como a semântica da instrução **proceed** é executar o código da hachura, esses fragmentos de código são extraídos dos seus locais originais para novos métodos, chamados de métodos de implementação de

```
public void setX(int);
  Code:
   0:   invokestatic    #62; //Method TestAspect.aspectOf:()LTestAspect;
   3:   invokevirtual   #65; //Method TestAspect.
      ajc$before$TestAspect$1$887a4880:()V
   6:   aload_0 //Carga do objeto this
   7:   iload_1 //Carga do primeiro parâmetro
   8:   putfield    #29; //Field x:I
   11:  return
```

Listagem 2.11: Adendo anterior aplicado à execução de setX.

```
after(): call(void Figure.setX(int)) {
    System.out.println("just called setX");
}
```

Listagem 2.12: Adendo posterior declarado em TestAspect.

```
public static void main(java.lang.String[]);
  Code:
   /* ... */
   13:  bipush  30
   15:  invokevirtual   #33; //Method Circle.setX:(I)V
   18:  goto    30
   21:  astore_2
   22:  invokestatic    #60; //Method TestAspect.aspectOf:()LTestAspect;
   25:  invokevirtual   #63; //Method TestAspect.
      ajc$after$TestAspect$1$d147c5c6:()V
   28:  aload_2
   29:  athrow
   30:  nop
   31:  invokestatic    #60; //Method TestAspect.aspectOf:()LTestAspect;
   34:  invokevirtual   #63; //Method TestAspect.
      ajc$after$TestAspect$1$d147c5c6:()V
   /* ... */
  Exception table:
   from    to  target type
    15    18    21   Class java/lang/Throwable
     0    41    41   Class java/lang/Exception
```

Listagem 2.13: Aplicação do adendo da Listagem 2.12.

hachuras (*shadow methods*). Quaisquer variáveis do contexto, ou ambiente, original da hachura necessárias para sua execução são passadas como argumentos para o método para onde a hachura é extraída – passagem de contexto na costura de adendos é descrita em detalhes na Seção 2.2.1.3.

O adendo da Listagem 2.14 substitui chamadas ao método setX da classe Figure e apenas prossegue para a execução de suas hachuras. O código criado na costura desse adendo, setX_aroundBody1$advice, é mostrado na Listagem 2.15. No código do método main, uma chamada ao adendo substitui completamente a chamada a setX, ao contrário do que ocorria na costura dos adendos apresentados anteriormente. Essa chamada encontra-se no método setX_aroundBody0, criado para conter a hachura. A implementação da hachura é chamada no corpo da expansão do adendo, em lugar do comando **proceed**.

Quando instruções **proceed** são usadas em classes anônimas aninhadas no corpo de adendos de contorno, o combinador substitui o código da hachura pela instanciação de uma implementação do tipo AroundClosure. Como descrito em [HH04], essa interface de *closure* é simplesmente uma abstração da implementação de código de hachura.

Chamadas a **proceed** que aparecem em tipos aninhados no corpo de adendos de contorno são colocadas em objetos de *closure*, pois elas estão fechadas no seu escopo. Isso significa que essas chamadas em tipos aninhados podem acessar variáveis que estão no escopo do adendo, mesmo quando esse escopo não estiver mais disponível. Quando *ajc* costura adendos de contorno que possuem chamadas **proceed** em tipos aninhados, esses valores são associados a objetos *closure* no lugar de chamadas a **proceed** no corpo do adendo, para que esse comando execute corretamente mesmo após o fim da execução do adendo.

Por exemplo, o adendo da Listagem 2.16 faz com que chamadas ao método setX da classe Figure sejam redirecionadas para uma nova linha de execução. Para isso cria-se, no seu corpo, uma especialização anônima de Thread, cujo método run contém a chamada a **proceed**. Como nos exemplos anteriores de adendos de contorno, a costura do adendo no método main gera um método setX_aroundBody0 contendo sua hachura, a chamada a setX.

Listagem 2.17 mostra o construtor e o método run do *closure* criado para conter a hachura – uma classe chamada Figs$AjcClosure1. O arranjo de objetos que seu construtor recebe contém as variáveis do escopo da hachura e do adendo que são usados na chamada **proceed**. Nesse exemplo, estão no arranjo de variáveis ligadas ao *closure* a instância de Circle usada para realizar a chamada a setX (linha 33 da Listagem 2.6) e o argumento dessa chamada.

No código *bytecode* gerado para a costura desse adendo no método main, mostrado na Listagem 2.18, constrói-se o arranjo de argumentos para o objeto *closure*. Por razões

```
void around(): call(void Figure.setX(int)) {
    proceed();
}
```

Listagem 2.14: Adendo de contorno aplicado a chamadas a Circle.setRadius.

```
public static void main(java.lang.String[]);
  Code:
   /* ... */
   13:  bipush  30
   15:  istore_2
   16:  astore_3
   17:  aload_3
   18:  iload_2
   19:  invokestatic    #67; //Method TestAspect.aspectOf:()LTestAspect;
   22:  aconst_null
   23:  invokestatic    #71; //Method setX_aroundBody1$advice:(LCircle;
      ILTestAspect;Lorg/aspectj/runtime/internal/AroundClosure;)V
   /* ... */
private static final void setX_aroundBody0(Circle, int);
  Code:
   0:   aload_0
   1:   iload_1
   2:   invokevirtual   #33; //Method Circle.setX:(I)V
   5:   return
private static final void setX_aroundBody1$advice(Circle, int, TestAspect
   , org.aspectj.runtime.internal.AroundClosure);
  Code:
   0:   aload_3
   1:   astore  4
   3:   aload_0
   4:   iload_1
   5:   invokestatic    #73; //Method setX_aroundBody0:(LCircle;I)V
   8:   return
```

Listagem 2.15: *Bytecode* de uma aplicação do adendo definido na Listagem 2.14.

```
void around(): call(void Figure.setX(int)) {
    new Thread() {
        public void run() {
            proceed();
        }
    }.start();
}
```

Listagem 2.16: Adendo de contorno com chamada **proceed** em tipo aninhado.

```
public Figs$AjcClosure1(java.lang.Object[]);
  Code:
   0: aload_0
   1: aload_1
   2: invokespecial #10 //Method org/aspectj/runtime/internal/
      AroundClosure."<init>":([Ljava/lang/Object;)V
   5: return
public java.lang.Object run(java.lang.Object[]);
  Code:
   0:   aload_0
   1:   getfield    #14; //Field org/aspectj/runtime/internal/
      AroundClosure.state:[Ljava/lang/Object;
   4:   astore_2
   5:   aload_2
   6:   bipush  0
   8:   aaload
   9:   checkcast   #16; //class Circle
   12:  aload_2
   13:  bipush  1
   15:  aaload
   16:  invokestatic    #22; //Method org/aspectj/runtime/internal/
      Conversions.intValue:(Ljava/lang/Object;)I
   19:  invokestatic    #28; //Method Figs.setX_aroundBody0:(LCircle;I)V
   22:  aconst_null
   23:  areturn
```

Listagem 2.17: *Closure* criado para conter a chamada proceed da Listagem 2.16.

de espaço ocultou-se o código que constrói esse arranjo, representado na listagem pelo comentário antes da instrução 42. Todavia, o arranjo é armazenado na posição 4 do *frame*, e carregado, pela instrução na posição 46, para a chamada ao construtor do *closure*. O método chamado pela instrução invokevirtual #78 é a implementação do adendo, que recebe o objeto *closure* por parâmetro[4].

```
public static void main(java.lang.String[]);
  Code:
   /* ... */
    42: new #65; //class Figs$AjcClosure1
   45:  dup
   46:  aload   4
   48:  invokespecial   #68; //Method Figs$AjcClosure1."<init>":([Ljava/
      lang/Object;)V
   51:  invokevirtual   #78; //Method TestAspect.
      ajc$around$TestAspect$1$d147c5c6:(Lorg/aspectj/runtime/internal/
      AroundClosure;)V
   /* ... */
```

Listagem 2.18: Aplicação do adendo da Listagem 2.16 em Figs.main.

Listagem 2.19 mostra os métodos do aspecto TestAspect que implementam o adendo de contorno e a chamada **proceed**. Ao contrário dos casos simples de adendos de contorno, como o exemplo da Listagem 2.14, o corpo do adendo não possui uma chamada ao método que implementa a hachura. Essa implementação do adendo apenas cria uma instância da classe aninhada TestAspect$1 e chama seu método start. O código *bytecode* do método run de TestAspect$1 é mostrado na Listagem 2.20. Note que no método run dessa classe é que se chama a implementação da hachura, que, por sua vez, chama o método run do *closure*, executando assim a hachura em uma nova linha de execução[5].

Como se pode observar pelo *bytecode* gerado para o pequeno exemplo da Listagem 2.16, a costura de adendos usando *closures* para chamadas **proceed** em tipos aninhados é mais complexa do que para adendos de contorno mais simples, que contém chamadas **proceed** em seu próprio corpo. A construção de objetos *closure*, durante a execução de programas AspectJ compilados por *ajc*, pode ter um grande impacto em desempenho, como evidenciado em [DGH+04]. A estratégia de costura para adendos

[4] A instrução dup dessa listagem duplica na pilha o objeto *closure* recém-criado, de forma que, após o consumo de sua primeira cópia para a chamada ao construtor, feito pela instrução invokespecial #68, ainda há uma cópia no topo da pilha, que serve como argumento para a chamada ao adendo.

[5] O método run da classe TestAspect$1 é a implementação de uma *thread*, e não deve ser confundido com o método run do tipo AroundClosure, definido em Figs$AjcClosure1.

```
public void ajc$around$TestAspect$1$d147c5c6(org.aspectj.runtime.internal
    .AroundClosure);
  Code:
    0:   new #31; //class TestAspect$1
    3:   dup
    4:   aload_0
    5:   aload_1
    6:   invokespecial   #34; //Method TestAspect$1."<init>":(LTestAspect;
         Lorg/aspectj/runtime/internal/AroundClosure;)V
    9:   invokevirtual   #37; //Method TestAspect$1.start:()V
    12:  return
static void ajc$around$TestAspect$1$d147c5c6proceed(org.aspectj.runtime.
    internal.AroundClosure)    throws java.lang.Throwable;
  Code:
    0:   aload_0
    1:   iconst_0
    2:   anewarray   #4; //class java/lang/Object
    5:   invokevirtual   #48; //Method org/aspectj/runtime/internal/
         AroundClosure.run:([Ljava/lang/Object;)Ljava/lang/Object;
    8:   invokestatic    #54; //Method org/aspectj/runtime/internal/
         Conversions.voidValue:(Ljava/lang/Object;)Ljava/lang/Object;
    11:  return
```

Listagem 2.19: Implementação de adendo e do comando proceed usando *closures*.

```
public void run();
  Code:
    0:   aload_0
    1:   getfield    #19; //Field val$ajc_aroundClosure:Lorg/aspectj/
         runtime/internal/AroundClosure;
    4:   invokestatic    #31; //Method TestAspect.
         ajc$around$TestAspect$1$d147c5c6proceed:(Lorg/aspectj/runtime/
         internal/AroundClosure;)V
    7:   return
```

Listagem 2.20: Implementação da subclasse de Thread aninhada no corpo do adendo.

de contorno adotada pelo compilador *abc* tem o objetivo de eliminar esse problema e é, todavia, mais simples do que a adotada por *ajc*.

2.2.1.2 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 experimentos apresentados em [DGH+04] mostram que instanciação de *closures* pode causar um grande impacto no desempenho de 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 mesmo adendo em uma 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 no código *bytecode* criado para representar o 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 compilação, por um processo de expansão (*inlining*). Esse passo adicional na costura realizado pelo *abc* reduz o custo da execução da instrução *proceed*. Nesta seção, portanto, não será apresentado código que utiliza diretamente os identificadores inteiros para hachuras, pois essa estratégia, no *bytecode* final gerado, é obsoleta. Todavia, tal código ainda é gerado pelo *abc* em etapas intermediárias da costura, antes da expansão.

Suponha que o programa da Listagem 2.6 (pág. 24), junto com o adendo definido na Listagem 2.14 (pág. 30), seja compilado com o *abc*[6]. A costura desse adendo cria uma expansão do seu corpo para cada aplicação que aparece no programa. Essas expansões são criadas no código *bytecode* da classe correspondente ao aspecto que declarou o

[6] Como nos exemplos anteriormente apresentados nessa seção, o adendo da Listagem 2.14 é declarado em um aspecto chamado TestAspect, que não define nenhum outro adendo.

adendo. Listagem 2.21 mostra o código *bytecode* do método main costurado, que inclui uma chamada à expansão do adendo inline1around$0.

No código *bytecode* compilado sem aspectos para o método Figs.main, mostrado na Listagem 2.7 (pág. 25), a instrução na posição 8 chama o construtor da classe Circle, e as instruções das posições 11, 13 e 15 realizam a chamada a setX, que é capturada pelo adendo. No *bytecode* costurado pelo *abc* da Listagem 2.21, após a chamada ao construtor da classe Circle, chama-se uma expansão do adendo definido em TestAspect. Note que o argumento da chamada a setX não é passado para essa expansão do adendo.

Listagem 2.22 mostra o código *bytecode* gerado para essa expansão. A segunda instrução dessa implementação do adendo é específica para a hachura contida no método main, pois o argumento passado para setX é carregado diretamente no seu corpo. Tal especificidade é obtida por meio de propagação de constantes, por meio da qual o *abc* detecta que a chamada a setX realizada nesse ponto do código sempre tem o argumento constante 30, e, propagando essa constante, cria uma expansão do adendo definido na Listagem 2.14 que sempre chama setX com esse valor como argumento.

A chamada a setX no construtor da classe Figure é outra hachura do adendo definido na Listagem 2.14. Nessa hachura, a propagação de constantes é incapaz de detectar um valor constante a ser passado para setX. Assim, a expansão do adendo criada para essa aplicação recebe, como valor de contexto, o argumento a ser passado para setX. O código *bytecode* dessa implementação, e da sua ativação no construtor da classe Figure, é mostrado na Listagem 2.23. O argumento a ser passado para a hachura do adendo é carregado, na chamada a inline0around$0, pela instrução iload_1, que carrega o primeiro parâmetro do construtor.

Quando *closures* são necessários para implementar adendos de contorno que possuem chamadas **proceed** em tipos aninhados, *abc* faz com que classes que contenham hachuras desse adendo implementem uma interface AroundClosure. Essa interface define um método para onde as hachuras são movidas. A vantagem dessa abordagem, quando comparada à adotada pelo *ajc*, é que, como a própria classe que contém as hachuras é uma implementação de *closure*, não é necessário criar novos objetos em todas as ativações de adendos. Essa estratégia é descrita em maiores detalhes em [Kuz04].

Embora essa estratégia seja usada durante a costura, código gerado pelo *abc* não possui objetos *closure* adicionais. A expansão de adendos, realizada após a costura, elimina a passagem de objetos desse tipo para adendos ao criar uma implementação específica do adendo para cada uma de suas aplicações, tornando o código gerado mais eficiente em tempo de execução e uso de memória.

No código *bytecode* final gerado pelo *abc* para costura de adendos de contorno com chamadas **proceed** em tipos aninhados, nenhum objeto *closure* é criado. Na classe criada para implementar o tipo aninhado, introduzem-se campos que correspondem

```
public static void main(java.lang.String[]);
  Code:
   /* ... */
   8:   invokespecial   #36; //Method Circle."<init>":(III)V
   11:  invokestatic    #9; //Method TestAspect.aspectOf:()LTestAspect;
   14:  pop
   15:  invokestatic    #21; //Method TestAspect.inline$1$around$0:(
      LCircle;)V
   /* ... */
```

Listagem 2.21: Aplicação do adendo de contorno da Listagem 2.14 ao método Figs.main costurada pelo *abc*.

```
public static final void inline$1$around$0(Circle);
  Code:
   0:   aload_0
   1:   bipush  30
   3:   invokevirtual   #38; //Method Circle.setX:(I)V
   6:   return
```

Listagem 2.22: *Bytecode* da aplicação do adendo definido na Listagem 2.14 ao corpo do método main.

```
public Figure(int, int);
  Code:
   0:   aload_0
   1:   invokespecial   #47; //Method java/lang/Object."<init>":()V
   4:   invokestatic    #11; //Method TestAspect.aspectOf:()LTestAspect;
   7:   pop
   8:   iload_1
   9:   aload_0
   10:  invokestatic    #3; //Method TestAspect.inline$0$around$0:(
      ILFigure;)V
   /* ... */
public static final void inline$0$around$0(int, Figure);
  Code:
   0:   aload_1
   1:   iload_0
   2:   invokevirtual   #41; //Method Figure.setX:(I)V
   5:   return
```

Listagem 2.23: *Bytecode* da aplicação do adendo definido na Listagem 2.14 ao corpo do construtor da classe Figure.

às variáveis de contexto usadas na chamada **proceed** fechada nesse tipo. Assim, em lugar da criação do tipo aninhado no corpo do adendo, associam-se aos seus campos os valores de contexto necessários para a execução da chamada **proceed** que pode então realizar-se mesmo após o fim do escopo do adendo.

Quando o adendo da Listagem 2.16 (pág. 31) é compilado junto ao exemplo desta seção, TestAspect$1 é a classe criada para implementar o tipo anônimo declarado no seu corpo. O código *bytecode* do método run, definido na sua declaração no corpo do adendo, é mostrado na Listagem 2.24.

```
public void run();
  Code:
   0:   aload_0
   1:   getfield      #34; //Field contextField3:I
   4:   aload_0
   5:   getfield      #43; //Field contextField15:I
   8:   istore_1
   9:   aload_0
   10:  getfield      #22; //Field contextField16:Ljava/lang/Object;
   13:  astore_0
   14:  tableswitch{ //1 to 2
        1: 36;
        2: 44;
        default: 52 }
   36:  iload_1
   37:  aload_0
   38:  invokestatic      #46; //Method Figure.
      abc$static$proceed$TestAspect$around$0:(ILjava/lang/Object;)V
   41:  goto      60
   44:  iload_1
   45:  aload_0
   46:  invokestatic      #27; //Method Figs.
      abc$static$proceed$TestAspect$around$0:(ILjava/lang/Object;)V
   49:  goto      60
   52:  new #33; //class java/lang/RuntimeException
   55:  dup
   56:  invokespecial     #25; //Method java/lang/RuntimeException."<init
      >":()V
   59:  athrow
   60:  return
```

Listagem 2.24: *Bytecode* do método run do tipo aninhado TestAspect$1.

No corpo desse método, carregam-se os campos de TestAspect$1 que foram associados a valores de contexto do corpo do adendo. A variável contextField3 é carregada pela instrução da posição 1, e usada em um *switch*, na posição 14. Essa variável armazena o identificador *classID*, que determina qual classe contém a implementação da hachura a

ser executada. Quando o adendo é aplicado à hachura no método Figs.main, executa-se o trecho de código a partir da posição 44, que chama a implementação da hachura expandida na classe Figs. Se o valor do *classID* for 1, o trecho a partir da posição 36 é executado, chamando a hachura definida na classe Figure.

As instruções nas posições 5 a 10 da Listagem 2.24 as variáveis capturadas do contexto das hachuras. Essas variáveis são associadas aos campos contextField15 e contextField16, introduzidas nas classes Figs e Figure, que contêm hachuras desse adendo. As variáveis de contexto são usadas nas chamadas às implementações das hachuras das instruções 38 e 46. Listagens 2.25 e 2.26 mostram o código *bytecode* desses métodos.

Implementações expandidas do adendo também são criadas para esse tipo de adendo de contorno. Listagem 2.27 mostra o código *bytecode* de uma das expansões criadas para esse exemplo. Assim como na costura do adendo sem chamadas **proceed** em tipos aninhados apresentado anteriormente nesta seção, uma implementação especial do adendo é criada para sua aplicação no corpo de Figs.main: o método inline0around$0 não recebe com parâmetro o argumento inteiro da chamada a setX. Na implementação especial, a instrução bipush 30, que coloca a constante 30 no topo da pilha, aparece no lugar do carregamento desse parâmetro, que aparece na posição 15 da Listagem 2.27.

Nota-se ainda, no código *bytecode* da expansão de adendo mostrada na Listagem 2.27, a associação de variáveis locais do adendo a campos de um objeto do tipo TestAspect$1, que permite a execução do seu método run independentemente do tempo de vida do adendo. O valor constante 1, carregado pela instrução iconst_1 da posição 25, é armazenado no campo contextField3 da instância de TestAspect$1, indicando que a hachura a ser executada é a definida na classe Figure, mostrada na Listagem 2.26.

```
public static final void shadow$18(int, Circle);
  Code:
    0:   aload_1
    1:   iload_0
    2:   invokevirtual   #32; //Method Circle.setX:(I)V
    5:   return
public static void abc$static$proceed$TestAspect$around$0(int, java.lang.
    Object);
  Code:
    0:   iload_0
    1:   aload_1
    2:   checkcast   #19; //class Circle
    5:   invokestatic    #29; //Method shadow$18:(ILCircle;)V
    8:   return
```

Listagem 2.25: Implementação de hachura na classe Figs.

```
public static final void shadow$14(int, Figure);
  Code:
    0:   aload_1
    1:   iload_0
    2:   invokevirtual   #48; //Method setX:(I)V
    5:   return
public static void abc$static$proceed$TestAspect$around$0(int, java.lang.
    Object);
  Code:
    0:   iload_0
    1:   aload_1
    2:   checkcast   #26; //class Figure
    5:   invokestatic    #36; //Method shadow$14:(ILFigure;)V
    8:   return
```

Listagem 2.26: Implementação de hachura na classe Figure

```
public static final void inline$1$around$0(TestAspect, int, Figure);
  Code:
   0:   new #37; //class TestAspect$1
   3:   astore_3
   4:   aload_3
   5:   aload_0
   6:   invokespecial   #59; //Method TestAspect$1."<init>":(LTestAspect
        ;)V
   9:   aload_3
   10:  aload_2
   11:  putfield    #33; //Field TestAspect$1.contextField16:Ljava/lang/
        Object;
   14:  aload_3
   15:  iload_1
   16:  putfield    #65; //Field TestAspect$1.contextField15:I
   19:  aload_3
   20:  iconst_0
   21:  putfield    #51; //Field TestAspect$1.contextField4:I
   24:  aload_3
   25:  iconst_1
   26:  putfield    #52; //Field TestAspect$1.contextField3:I
   29:  aload_3
   30:  iconst_0
   31:  putfield    #50; //Field TestAspect$1.contextField2:I
   34:  aload_3
   35:  aconst_null
   36:  putfield    #14; //Field TestAspect$1.contextField1:
        LAbc$proceed$TestAspect$around$0;
   39:  aload_3
   40:  invokevirtual   #36; //Method TestAspect$1.start:()V
   43:  return
```

Listagem 2.27: *Bytecode* da expansão do adendo que implementa sua aplicação no corpo do método main da classe Figs.

2.2.1.3 Passagem de Contexto

Tanto na estratégia de costura adotada pelo *ajc* quanto pelo *abc*, a descrição deste texto ignora 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 variáveis de contexto na expressão de um conjunto de junção e usá-las no corpo de um adendo, e na instrução **proceed**. Variáveis de contexto devem ser passadas, portanto, 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].

Suponha que o adendo da Listagem 2.28, que captura argumentos de chamadas a setX, seja acrescentado ao programa apresentado anteriormente nesta seção. O código *bytecode* gerado para a hachura desse adendo no corpo de main é mostrado na Listagem 2.29.

```
before(int x): call(void Figure+.setX(int)) && args(x) {
    System.out.println("calling setX with arg " + x);
}
```

Listagem 2.28: Adendo anterior que captura variáveis de contexto de suas hachuras.

```
public static void main(java.lang.String[]);
  Code:
   /* ... */
   13:  bipush  30
   15:  istore_2 //Armazena variável de contexto no frame
   16:  invokestatic    #60; //Method TestAspect.aspectOf:()LTestAspect;
   19:  iload_2 //Carrega variável de contexto para chamada ao adendo
   20:  invokevirtual   #63; //Method TestAspect.
      ajc$before$TestAspect$1$e1c5f4fb:(I)V
   23:  iload_2
   24:  invokevirtual   #33; //Method Circle.setX:(I)V
   /* ... */
```

Listagem 2.29: Código *bytecode* de uma aplicação do adendo da Listagem 2.28.

Passagem de contexto para adendos de contorno é mais complexa do que para

os outros tipos de adendo, pois hachuras de adendos desse tipo são extraídas de seu ponto no código e podem ser ativadas por meio do comando **proceed**. Assim, variáveis de contexto capturadas em adendos de contorno devem estar disponíveis não só no corpo do adendo, mas precisam também ser passadas para as implementações de suas hachuras. Se o adendo da Listagem 2.30 for acrescentado ao programa desta seção, *ajc* gerará na sua costura o código *bytecode* mostrado na Listagem 2.31[7].

No código gerado para o método setX_aroundBody1$advice mostrado na Listagem 2.31, a chamada a setX_aroundBody0 da posição 32 corresponde à implementação do comando **proceed** no corpo do adendo definido na Listagem 2.30. Note que a lista de parâmetros da assinatura desse método, que implementa a hachura interceptada, é composta das variáveis necessárias para sua execução.

```
void around(int x): call(void Figure+.setX(int)) && args(x) {
    System.out.println("calling setX with arg " + x);
    proceed(x);
    System.out.println("just called setX");
}
```

Listagem 2.30: Adendo de contorno que captura uma variável de contexto.

2.2.1.4 Conjuntos de Junção do Tipo *cflow*

No escopo de costura de adendos em AspectJ, é necessário também considerar os 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. Essa cláusula oferece 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.

[7] Por questões de objetividade, omitiu-se nessa listagem o código *bytecode* modificado do método main, onde a chamada a setX é substituída por uma chamada a setX_aroundBody1$advice

```
private static final void setX_aroundBody0(Circle, int);
  Code:
   0:   aload_0
   1:   iload_1
   2:   invokevirtual   #33; //Method Circle.setX:(I)V
   5:   return
private static final void setX_aroundBody1$advice(Circle, int, TestAspect
   , int, org.aspectj.runtime.internal.AroundClosure);
  Code:
   /* ... */
   29:  aload_0
   30:  iload   6 //Carrega variável de contexto para chamada ao adendo
   32:  invokestatic    #89; //Method setX_aroundBody0:(LCircle;I)V
   35:  getstatic   #13; //Field java/lang/System.out:Ljava/io/
      PrintStream;
   38:  ldc #79; //String "just called setX"
   40:  invokevirtual   #41; //Method java/io/PrintStream.println:(Ljava/
      lang/String;)V
   43:  return
```

Listagem 2.31: *Bytecode* de uma aplicação do adendo da Listagem 2.30.

Um conjunto de junção da forma **cflow**(p) seleciona pontos de junção que ocorrem sob o fluxo de controle dos pontos selecionados pelo conjunto de junção 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.

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 usam as cláusulas **args**, **target** ou **this**, 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 a chamadas a foo, o combi-

nador 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.

As otimizações intra-procedurais 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 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. afirmam, em [ACH+05b], que existem dois tipos de hachuras associados a cada conjunto de junção **cflow** que causam perda de desempenho na execução do programa: uma hachura de atualização (*update shadow*) e uma hachura de consulta (*query shadow*).

Para um dado conjunto de junção da forma 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 que casam com q na execução do programa onde a existência de pontos de junção q na pilha de chamadas. Na expressão de **cflow** foobar da página 43, hachuras de consulta são chamadas ao método foo, e hachuras de atualização são chamadas ao método bar. 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 possível determinar estaticamente 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 relacionados à construção **cflow** em tempo de compilação, como mostrado em [ACH+05b], tornando programas que dependem de conjuntos de junção do tipo **cflow** mais eficientes.

2.3 Arquitetura e Objetivos do Compilador *ajc*

O compilador *ajc* [Aspb] é a implementação oficial da linguagem AspectJ. Ele é implementado em Java, como uma extensão do compilador do projeto *Eclipse Java Development Tools*[8] (JDT). O compilador JDT é incremental, ou seja, é capaz de incorporar novos trechos de código a programas previamente compilados sem, para isso, repetir as etapas de compilação já realizadas sobre o código original. Além disso, esse compilador é capaz de gerar programas executáveis mesmo a partir de código que contenha erros, ignorando os trechos com erros para que sejam compilados apenas em iterações seguintes do processo incremental. Tal característica é desejável, por exemplo, para a construção de compiladores integrados a ambientes de desenvolvimento, que sejam capazes de compilar código à medida em que ele é escrito pelo programador.

O compilador JDT também é extensível, por meio de adaptadores de comportamento, permitindo alterar a forma pela qual programas em Java são compilados. Adaptadores são invocados pelo compilador à medida que etapas da compilação são alcançadas e terminadas. O compilador *ajc* faz proveito desse arcabouço de extensão da compilação de programas escritos na linguagem Java para realizar a costura de código de aspectos em classes durante a compilação.

Hilsdale e Hugunin, em [HH04], descrevem as estratégias adotadas pelo compilador *ajc* durante a costura de adendos. Há ainda um documento [Aspa], disponível com o código-fonte desse compilador, que descreve a implementação de AspectJ realizada no compilador *ajc*. Entretanto, a porção desse documento que explica os trechos de código que correspondem ao casamento de pontos de junção e à costura de adendos é apresentada como um exemplo, em forma de receita, da implementação de uma nova construção para a linguagem AspectJ. Para a realização deste trabalho, entretanto,

[8] http://www.eclipse.org/jdt

foi necessário estudar em maior detalhe o código-fonte do compilador *ajc* de forma a compreender sua estrutura e nela identificar pontos de extensão, ou ganchos, que permitam implementar as otimizações propostas nos capítulos seguintes.

A extensão do compilador JDT para AspectJ usada no *ajc* introduz na compilação construções específicas dessa linguagem. Arquivos que definem classes Java e aspectos são tratados pelo *front-end* do compilador como unidades de compilação. O *ajc* identifica e processa todas as unidades de compilação, realizando análises léxica, sintática e semântica, análises de fluxo e geração de código *bytecode*. Após o processamento de todas as unidades fornecidas pelo programador, o *back-end* do *ajc* realiza a costura de código, que efetivamente combina código gerado para aspectos a código gerado para classes Java.

Código *bytecode* é representado internamente no compilador *ajc* por meio da biblioteca BCEL [DvZH03], que fornece uma representação em Java de código *bytecode* e também recursos para manipular esse código. Essa biblioteca foi ampliada pelos desenvolvedores do compilador *ajc* para torná-la capaz de armazenar, durante a costura, referências para *modificadores de hachuras* (tradução livre de *shadow mungers*[9]) existentes em membros de classes e seqüências de instruções. A Versão 5 do *ajc* modifica a biblioteca BCEL, tornando-a capaz de lidar com as construções incorporadas a Java 5 que geram novas construções também no *bytecode*: polimorfismo paramétrico e anotações.

Um adaptador do compilador JDT implementado pelo *ajc* é responsável por, após a compilação, iniciar o processo de costura. Adaptadores são definidos por meio de uma interface no arcabouço de compilação JDT onde se definem métodos que funcionam como ganchos durante o processo de compilação, sendo chamados antes e após várias etapas do processo de compilação. Particularmente para o adaptador usado pelo compilador *ajc*, a definição do método afterCompiling permite disparar a costura de código.

Figura 2.2 ilustra resumidamente em um diagrama de colaboração UML os passos e objetos envolvidos na compilação de um programa em AspectJ pelo compilador *ajc*[10]. Nomes de classes e pacotes foram omitidos nesse diagrama em prol da legibi-

[9] O dicionário *Longman Dictionary of Contemporary English* define o termo *munging* como "o processo de mudar parte de um endereço de *email* quando se enviam mensagens a páginas na Internet, para que empresas não possam copiar esse endereço e enviar para ele mensagens indesejadas". No contexto de costura de código em linguagens orientadas por aspectos entende-se, por *munger*, uma entidade que modifica hachuras, tal como um adendo.

[10] Note que os diagramas apresentados nesta seção propositadamente ofuscam um grande número de detalhes do código do compilador *ajc*, buscando apresentar ao leitor apenas a interação em alto nível entre os objetos nele representados.

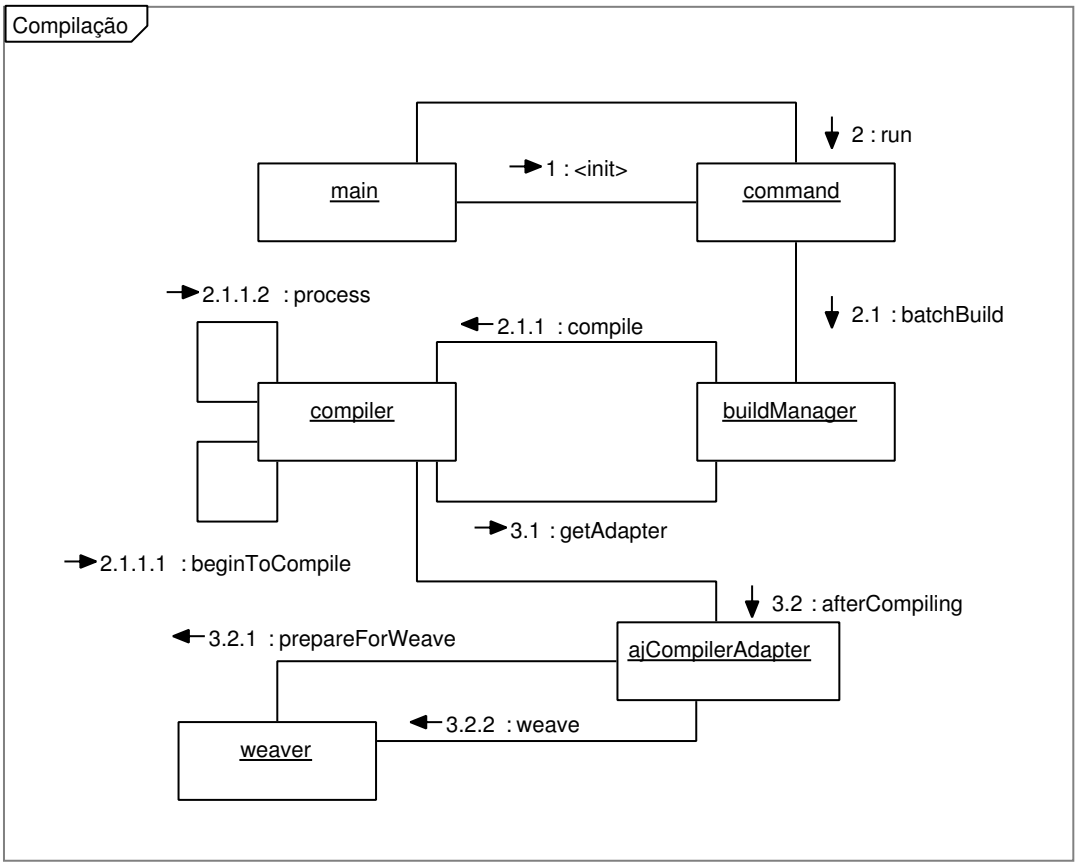

Figura 2.2: Diagrama de colaboração para a compilação de um programa pelo *ajc*.

lidade, e são descritos no texto. O objeto main[11] recebe os argumentos da linha de comando do compilador e os repassa durante a criação de um command[12]. Um gerenciador de compilação, buildManager[13], é então chamado para instanciar um compiler[14] e iniciar a compilação. O objeto compiler inicia, então, o processamento das unidades de compilação definidas na linha de comando onde o *ajc* foi chamado.

Uma vez terminada a compilação, buildManager é novamente invocado para instanciar o adaptador de compilação adequado para a costura, ajCompilerAdapter; a operação afterCompiling desse adaptador é então chamada, dando início ao processo de costura.

A costura de código do compilador *ajc* é implementada por uma estrutura complexa de hachuras (shadows), adendos (advices), um combinador (weaver) e combinadores de classes (classWeavers). Esses objetos e a interação entre eles, até o momento da inserção de fato de código nas classes costuradas, são mostrados no diagrama da Figura 2.3. Note

[11] Classe org.aspectj.tools.ajc.Main
[12] Classe org.aspectj.ajdt.AjdtCommand
[13] Classe org.aspectj.ajdt.internal.core.builder.AjBuildManager
[14] Classe org.aspectj.org.eclipse.jdt.internal.compiler.Compiler

que nesse diagrama o elemento classWeaver e os vários classWeaver∗ representam o mesmo objeto, apresentado por diferentes elementos do diagrama para melhor legibilidade; o mesmo ocorre para shadow e shadow∗.

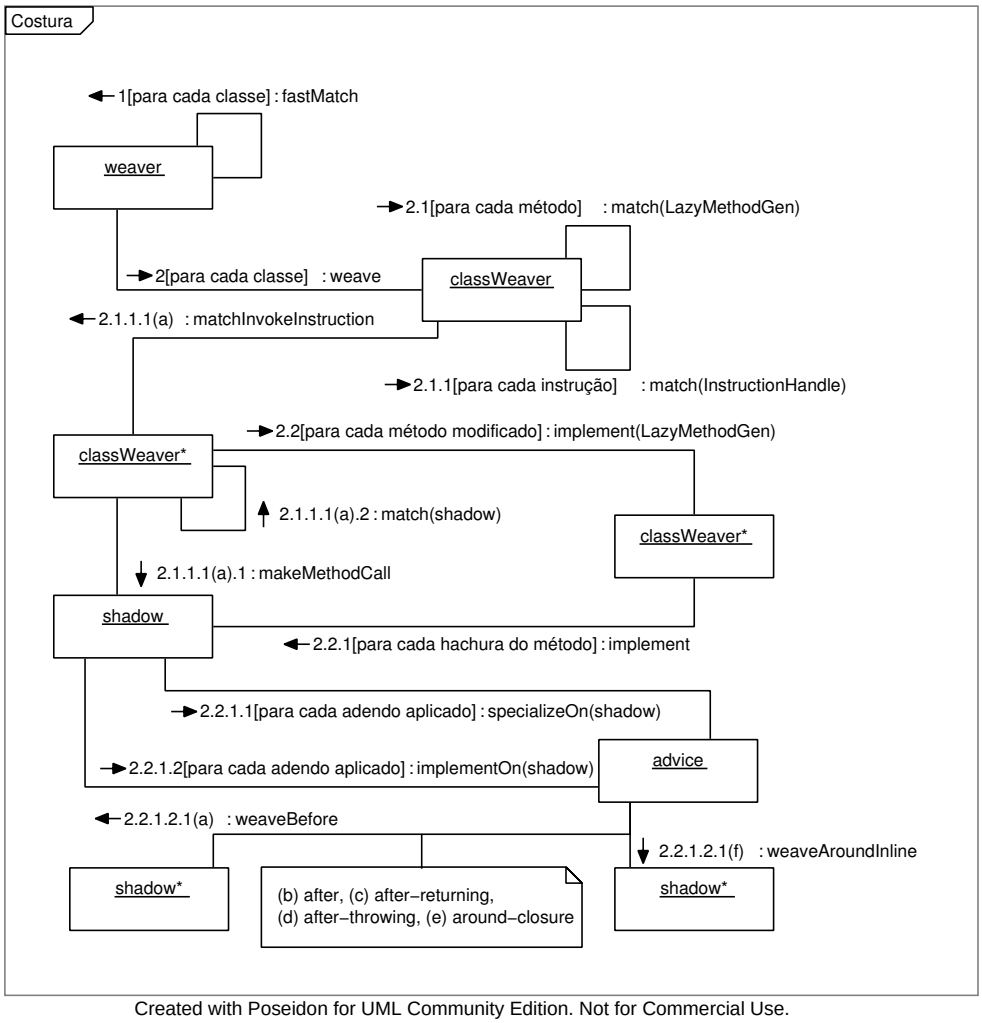

Figura 2.3: Diagrama de colaboração para a costura de código realizada pelo *ajc*.

O início da execução dessa colaboração pelo objeto weaver[15] decorre da chamada ao seu método weave realizada no fim da execução representada pelo diagrama da Figura 2.2. O weaver então casa adendos e classes, em fastMatch para identificar hachuras no código onde há introdução de comportamento implementado por aspectos. Para cada classe em que aspectos são aplicados, weaver cria um classWeaver[16] responsável por tratar sua costura de código. Cada classWeaver então percorre os métodos de sua respectiva classe identificando hachuras nas instruções do seu código *bytecode*.

[15] Classe org.aspectj.weaver.bcel.BcelWeaver
[16] Classe org.aspectj.weaver.bcel.BcelClassWeaver

A chamada 2.1.1.1(a) do diagrama da Figura 2.3 é realizada quando classWeaver encontra uma instrução de invocação, que, por definição, pode ser hachura de um adendo que se aplique a chamadas de método, ou seja, de adendos cujo conjunto de junção seja da forma **call**(m), onde m é uma expressão que determina o padrão de nomes de métodos cujas invocações são capturadas. Cria-se um shadow[17] para representar essa hachura, e chama-se classWeaver.match(shadow), que identifica se essa hachura de fato corresponde a algum adendo. Chamadas similares a 2.1.1.1(a), omitidas por legibilidade, criam hachuras para instruções que acessam atributos de classes para escrita e leitura, ou seja, hachuras de adendos cujo conjunto de junção são da forma **set**(f) ou **get**(f).

Em seguida, para cada método modificado pela execução de classWeaver.match da chamada 2.1, invoca-se, nas chamadas 2.2.1.1 e 2.2.1.2, a representação de um adendo, advice[18], para que seu código seja especializado para cada hachura e implementado na classe em que essa hachura ocorre. A inserção de fato de código de adendos nas classes que contêm hachuras é realizada pelas chamadas 2.2.1.2.1(a – f), dependendo do tipo de adendo e do seu corpo. Note que, na hierarquia do compilador *ajc*, a classe BcelAdvice é um subtipo de ShadowMunger, ou seja, adendos são também os objetos modificadores de hachuras.

Nesse ponto da costura, aspectos já foram transformados pelo *front-end* em classes comuns em representação BCEL de código *bytecode*, de forma que atributos e métodos desses aspectos são disponíveis como membros de suas classes.

Especialização de um adendo compreende tarefas de preparação para implementar a aplicação desse adendo a determinada hachura. Um exemplo de estrutura específica para cada hachura é o objeto **thisJoinPoint**, acessível no corpo de adendos, e que contém, durante a execução, informações sobre o ponto de junção capturado; para cada hachura do adendo, um objeto **thisJoinPoint** deve ser criado. Outra estrutura que precisa ser construída para cada hachura é a sua lista de argumentos, sejam eles variáveis presentes no contexto da hachura ou especificamente capturadas pela expressão do conjunto de junção usada pelo adendo.

Por exemplo, na implementação de um adendo anterior em uma hachura, uma chamada ao adendo é inserida no *bytecode* imediatamente antes da hachura, por meio da biblioteca BCEL. Para adendos de contorno que não contêm chamadas a **proceed** definidas em classes anônimas no corpo do adendo, a hachura é extraída de seu local original para um método e cria-se uma implementação local do adendo na classe em que a hachura ocorre. O corpo da implementação local consiste numa cópia do corpo do adendo implementado na classe que define seu aspecto, exceto pela instrução **proceed**,

[17] Classe org.aspectj.weaver.bcel.BcelShadow
[18] Classe org.aspectj.weaver.bcel.BcelAdvice

que é substituída por uma chamada ao método para onde se extraiu a hachura.

Observa-se novamente que, na costura de código, tanto classes quanto aspectos já foram transformados em código *bytecode*, de forma que a cópia do corpo do adendo é feita diretamente por meio da biblioteca BCEL, salvo devidos cuidados com a manutenção de consistência na pilha de execução.

Na costura de adendos de contorno, a implementação local do adendo e o método para onde a hachura é extraída são criados em weaveAroundInline. É exatamente essa criação que se deve evitar na ocorrência de várias hachuras idênticas em uma mesma classe, reaproveitando métodos já criados para reduzir o tamanho do código gerado.

2.4 Arquitetura e Objetivos do Compilador *abc*

O compilador *abc* é um ambiente de pesquisa, que implementa a linguagem AspectJ e permite experimentar novas construções e otimizações sobre essa linguagem. Os principais objetivos desse compilador são extensibilidade e geração de código eficiente. Para atingir esses objetivos, usa-se o *front-end* extensível Polyglot e, no *back-end*, o otimizador de código *bytecode* Soot [ACH+04].

Figura 2.4 mostra, em alto nível, o processo de compilação de programas AspectJ implementado pelo *abc*. O *front-end* baseado no Polyglot produz, a partir de código de classes Java e aspectos, uma árvore de sintaxe abstrata (AST) do programa. Essa árvore é então separada em uma AST do programa Java e um conjunto de informações sobre os aspectos do programa, chamado *AspectInfo*. Em seguida, gera-se código em uma representação intermediária, chamada *Jimple*, do programa Java, e os aspectos são costurados sobre essa representação. Após a costura, o código intermediário é otimizado e transformado em *bytecode*, que é o produto final da compilação.

Extensões do compilador *abc* são criadas por meio de uma classe AbcExtension, que pode definir modificações no *front-end* e no *back-end* do compilador. O *front-end* pode ser modificado por meio da alteração da gramática da linguagem AspectJ e da AST gerada durante o *parsing*. O *back-end* pode ser modificado por meio de novas otimizações, implementadas com o arcabouço Soot. Na execução do *abc*, ativam-se extensões por meio de uma marcação na linha de comandos.

A costura de código no *abc* é realizada sobre Jimple, uma representação intermediária de nível mais alto do que o *bytecode*. Essa representação, que consiste em um código de 3 endereços, não depende da máquina de pilha do código *bytecode*, o que, segundo os desenvolvedores do compilador [ACH+05a, ACH+04], simplifica a costura de código. Jimple é parte do arcabouço Soot de otimização, e otimizações definidas no compilador *abc* também operam sobre essa representação de programas.

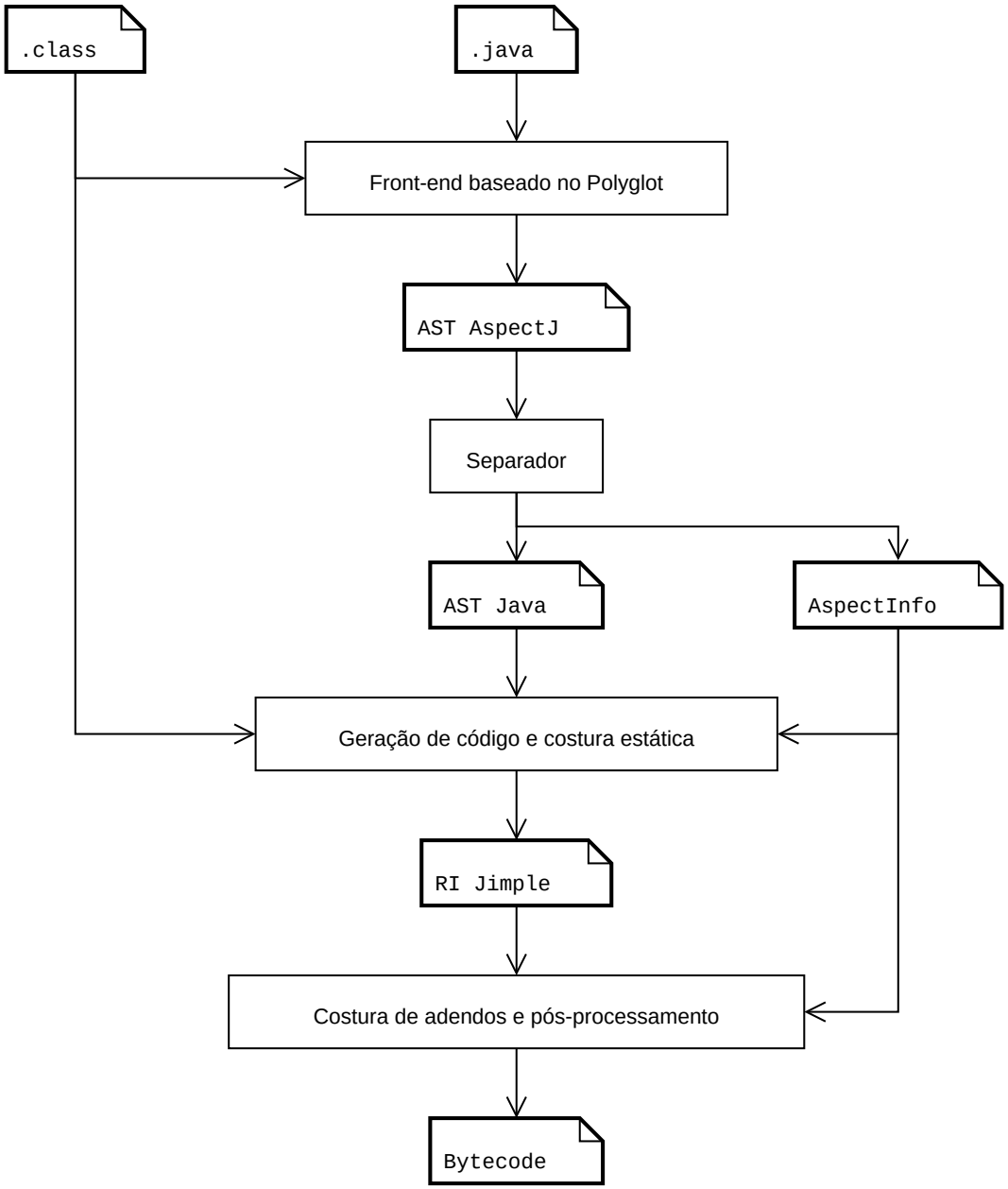

Figura 2.4: Arquitetura em alto nível do compilador *abc*. Extraída de [ACH+04]

2.5 Conclusões

Embora os compiladores *ajc* e *abc* tenham sido desenvolvidos com diferentes objetivos, ambos aplicam técnicas similares na compilação de programas em AspectJ. O estudo das estratégias de costura adotadas pelos dois compiladores possibilita a identificação de problemas em comum nessas estratégias, que podem ser considerados problemas da costura de código AspectJ em geral, e não apenas de uma implementação.

Outra contribuição obtida ao se estudar diferentes compiladores e a integração de otimizações a eles está em avaliar a extensibilidade de cada um por meio das dificulda-

des encontradas no processo de implementação. Nos próximos capítulos apresentam-se problemas identificados em código gerado por esses compiladores, e as soluções propostas para eles.

Capítulo 3

Caracterização do Problema

Compilação em linguagens orientadas por aspectos é um tema recente e de atual desenvolvimento. Embora poucos trabalhos na literatura descrevam em detalhes as técnicas aplicadas em tais compiladores [HH04, ACH+05a, ACH+05b], um estudo aprofundado de código gerado por eles permite identificar problemas e nichos de otimização não explorados.

Este capítulo relata problemas identificados em código gerado pelo compilador *ajc*, versão 1.5, e *abc*, versão 1.2.1. Esse problemas são abordados por meio de um programa AspectJ de exemplo, e trechos do código *bytecode* gerado para esse programa pelos compiladores *ajc* e *abc* são analisados para levantar os pontos de otimização explorados neste trabalho.

3.1 Repetição de Implementações de Adendos e Hachuras

A expansão de adendos de contorno realizada em compiladores da linguagem AspectJ durante a costura introduz no código *bytecode* de programas AspectJ implementações especializadas desses adendos. Para cada hachura h_i de um adendo de contorno a_j em uma classe C, cria-se no código dessa classe uma implementação local de a_j, e extrai-se h_i de seu local original para um novo método. A chamada a **proceed** no corpo da implementação local de a_j é substituída por uma chamada ao novo método que implementa h_i.

Esse processo é mostrado na Figura 3.1. Nessa figura, as caixas sombreadas são hachuras de adendos de contorno: $h1$ é uma hachura do adendo $a1$, e $h2$ e $h3$ são hachuras de $a2$. Após a costura de código, implementações de $a1$ e $a2$ são introduzidas no corpo da classe C, e os comandos **proceed** do corpo desses adendos são substituídos por chamadas às implementações extraídas de suas hachuras.

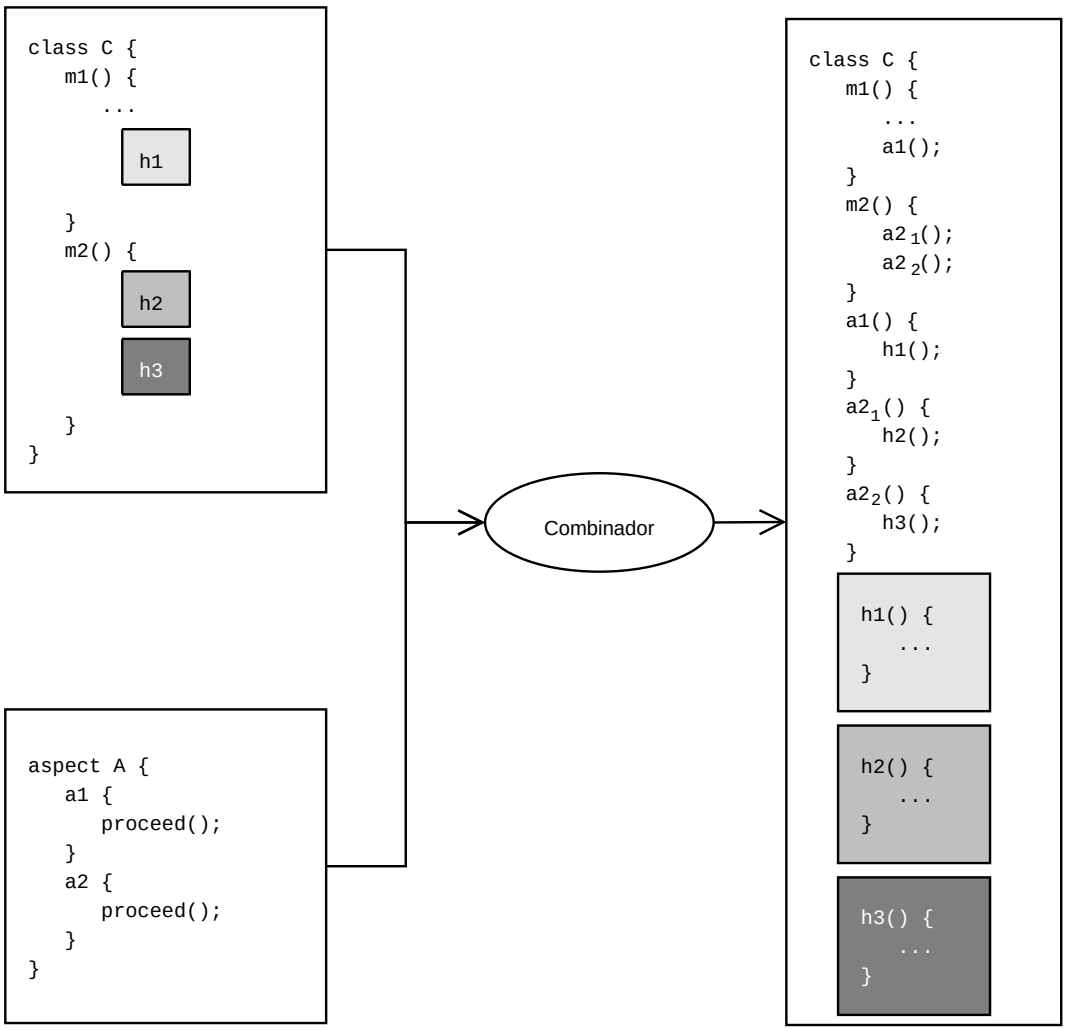

Figura 3.1: Expansão de adendos de contorno e hachuras.

Suponha, entretanto, que C possua diversas hachuras idênticas do adendo a; nesse caso, diversas implementações locais de a serão introduzidas na classe C, aumentando desnecessariamente o *bytecode* dessa classe. No exemplo da Figura 3.1, se o código das hachuras $h2$ e $h3$ forem iguais, os métodos $h2$ e $h3$ são semanticamente equivalentes, o que torna $a2_1$ e $a2_2$ também semanticamente equivalentes. O restante desta seção apresenta o problema de repetição de adendos e hachuras em maior detalhe, via um pequeno exemplo, cujo código gerado pelos compiladores *ajc* e *abc* é estudado.

Algumas das listagens desta seção possuem representações textuais de código *bytecode*. Embora uma descrição completa de sua estrutura e suas instruções esteja fora do escopo deste texto, pontos relevantes a esta discussão são apresentados aos poucos, junto aos trechos de código que os contêm.

Considere o código da Listagem 3.1. Classe Point define um ponto com dimensões x e y inteiras, e classe Main define métodos que criam uma instância de Point e manipulam as coordenadas do ponto representado. A saída esperada desse programa é

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

Suponha ainda que uma restrição seja introduzida na especificação do programa, proibindo a atribuição de valores negativos às coordenadas de pontos. Usando AOP, é possível isolar a implementação desse requisito, que abrange toda a aplicação, por meio do aspecto da Listagem 3.2 um aspecto, em vez de alterar os métodos afetados na classe Point[1]. Escreve-se, assim, o aspecto da Listagem 3.2.

O adendo de contorno definido em CoordCheckAspect garante que a restrição de "coordenadas não-negativas" seja respeitada, capturando e manipulando os argumentos para cada chamada a setPosition(**int**,**int**). Com esse aspecto, a saída do programa torna-se:

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

Descreve-se, a seguir, o código *bytecode* gerado pelos compiladores *ajc* e *abc* para este exemplo.

3.1.1 Exemplo Compilado

Como descrito no Capítulo 2, os compiladores *ajc* e *abc* adotam diferentes abordagens na costura de adendos de contorno. Nesta seção apresenta-se o código gerado por esses

[1] 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.

```
1  public class Point {
2      private int x;
3      private int y;
4      public Point(int x, int y) {
5          setPosition(x,y);
6      }
7      public int getX() { return x; }
8      public int getY() { return y; }
9      public void setPosition(int x, int y) {
10         this.x = x;
11         this.y = y;
12     }
13     public void moveBy(int dx, int dy) {
14         setPosition(this.x + dx, this.y + dy);
15     }
16     public String toString() {
17         return "(" + x + "," + y + ")";
18     }
19 }
20 public class Main {
21     public static void main(String[] args) {
22         Point p = new Point(0,0);
23         p.moveBy(14,27);
24         System.out.println(p);
25         doSomething(p);
26     }
27     private static void doSomething(Point p) {
28         p.setPosition(-5,5);
29         System.out.println(p);
30         p.moveBy(40,100);
31         System.out.println(p);
32     }
33 }
```

Listagem 3.1: Um pequeno programa que manipula pontos com duas dimensões inteiras.

dois compiladores para o exemplo das Listagens 3.1 e 3.2. Os problemas identificados em ambos são apresentados na Seção 3.1.2.

Estratégia Adotada pelo *ajc*

De acordo com a estratégia de costura de adendos de contorno descrita na Seção 2.2.1.1, 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 *bytecode* apresentado nesta seção foi gerado pelo compilador *ajc*, versão 1.5.2.

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

Listagem 3.2: Um aspecto para garantir que coordenadas de pontos não são negativas.

Listagem 3.3 mostra uma representação textual[2] do código *bytecode* do método doSomething da classe Main antes da introdução do aspecto CoordCheckAspect no programa. As instruções nos índices 0 a 3 carregam o objeto-alvo e os argumentos da chamada ao método setPosition realizada na instrução 4. Essas quatro instruções constituem a hachura do adendo de contorno declarado no aspecto introduzido por CoordCheckAspect, pois compõem uma chamada ao método setPosition da classe Point.

Essa hachura é extraída de seu local para um novo método, e seu código é mostrado na Listagem 3.4. O ambiente disponível no local original da chamada ao método setPosition é repassado pelo combinador para esse método, de modo que a semântica da hachura é mantida em seu novo local de implementação. As instruções de carga que aparecem nessa listagem obtêm valores passados como argumentos para a hachura do seu ambiente original.

Após gerar na classe Main uma expansão do adendo definido em Coord−CheckAspect (Listagem 3.2), o combinador substitui a chamada **proceed** no corpo dessa implementação por uma chamada a setPosition_aroundBody0. O código dessa expansão de adendo é mostrado na Listagem 3.5.

A mesma técnica de costura é aplicada às duas hachuras presentes, nas linhas 5 e 14 (veja Listagem 3.1, pág. 56), no código da classe Point. O código gerado na costura dessa classe é mostrado mais adiante em discussão oportuna.

Estratégia Adotada pelo *abc*

O código *bytecode* apresentado nesta seção foi gerado pelo compilador *abc*, versão 1.2.1. Exceto pelos valores das constantes associadas a métodos e campos, o código

[2] Representações textuais de código *bytecode*, como a da Listagem 3.3, podem ser obtidas por meio do programa *javap*, que faz parte do Java SDK.

```
private static void doSomething(Point);
  Code:
   0:   aload_0 //Carrega objeto-alvo
   1:   bipush  -5 //Primeiro argumento
   3:   iconst_5 //Segundo argumento
   4:   invokevirtual   #47; //Method Point.setPosition:(II)V
   7:   getstatic   #30; //Field java/lang/System.out:Ljava/io/
      PrintStream;
   10:  aload_0
   11:  invokevirtual   #36; //Method java/io/PrintStream.println:(Ljava/
      lang/Object;)V
   14:  aload_0
   15:  bipush  40
   17:  bipush  100
   19:  invokevirtual   #24; //Method Point.moveBy:(II)V
   22:  getstatic   #30; //Field java/lang/System.out:Ljava/io/
      PrintStream;
   25:  aload_0
   26:  invokevirtual   #36; //Method java/io/PrintStream.println:(Ljava/
      lang/Object;)V
   29:  return
```

Listagem 3.3: *Bytecode* do método doSomething da classe Main.

```
private static final void setPosition_aroundBody0(Point, int, int);
  Code:
   0:   aload_0 //Carrega objeto-alvo
   1:   iload_1 //Primeiro argumento
   2:   iload_2 //Segundo argumento
   3:   invokevirtual   #47; //Method Point.setPosition:(II)V
   6:   return
```

Listagem 3.4: Hachura de um adendo de contorno extraída para um método.

gerado pelo compilador *abc* para o método doSomething, antes da introdução do aspecto CoordCheckAspect, é idêntico ao da Listagem 3.3, gerado por *ajc*.

Implementações de adendos são criadas pelo combinador *abc*, não na classe em que suas hachuras ocorrem, mas sim nas classes que implementam aspectos em que os adendos foram definidos. Listagem 3.6 mostra o código *bytecode* gerado pelo *abc* para o método doSomething após a introdução de CoordCheckAspect no programa da Listagem 3.1. A chamada ao adendo, nessa listagem, é realizada na instrução 5.

No código dessa listagem, note que os parâmetros da chamada a setPosition , em contraste ao código gerado pelo *ajc*, não são passados para a implementação do adendo.

```
private static final void setPosition_aroundBody1$advice(Point, int, int,
    CoordCheckAspect, int, int, org.aspectj.runtime.internal.
   AroundClosure);
  Code:
   0:   iload   4 //Carrega parâmetro x
   2:   ifge    8 //Se x > 0, pula para instr. 8
   5:   iconst_0 //Senão,
   6:   istore  4 //faz x = 0
   8:   iload   5 //Carrega parâmetro y
   10:  ifge    16 //Se y > 0, pula para instr. 16
   13:  iconst_0 // Senão,
   14:  istore  5 //faz y = 0
   16:  iload   4 //Carrega x
   18:  iload   5 //Carrega y
   20:  aload   6 //Carrega closure
   22:  astore  7
   24:  istore  8
   26:  istore  9
   28:  aload_0 //Carrega objeto-alvo
   29:  iload   9 //Carrega x
   31:  iload   8 //Carrega y
   33:  invokestatic    #68; //Method setPosition_aroundBody0:(LPoint;II)V
   36:  return
```

Listagem 3.5: Implementação local do adendo definido em CoordCheckAspect.

O *back-end* do compilador *abc* realiza diversas otimizações sobre o *bytecode* gerado. Dentre essas otimizações está a propagação de constantes, que analisa o *bytecode* substituindo parâmetros por constantes, quando possível. O efeito dessa propagação, no *bytecode* do programa apresentado, é que a implementação do adendo de contorno aplicada ao método doSomething usa as constantes 0 e 5, pois esses são sempre os valores com que setPosition será chamado nessa aplicação do adendo (veja listagens 3.1 e 3.2).

Listagem 3.7 mostra a implementação da aplicação do adendo definido em Coord−CheckAspect ao método doSomething (inline2around$0) e moveBy (inline$1$around$0). Note que o compilador *abc* não cria um método para conter a hachura, substitui a chamada a **proceed** no corpo do adendo pela chamada ao método interceptado; nesse exemplo, como se pode observar na Listagem 3.7, chamadas ao método setPosition são inseridas diretamente no corpo de inline1around$0 e inline$2$around$0.

```
public static final void inline$2$around$0(Point);
  Code:
   0:   aload_0 //Carrega objeto-alvo
   1:   iconst_0 //Primeiro argumento
```

```
private static void doSomething(Point);
  Code:
   0:   invokestatic    #30; //Method CoordCheckAspect.aspectOf:()
      LCoordCheckAspect;
   3:   pop
   4:   aload_0
   5:   invokestatic    #15; //Method CoordCheckAspect.inline$2$around$0
      :(LPoint;)V
   8:   getstatic   #39; //Field java/lang/System.out:Ljava/io/
      PrintStream;
   11:  aload_0
   12:  invokevirtual   #16; //Method java/io/PrintStream.println:(Ljava/
      lang/Object;)V
   15:  aload_0
   16:  bipush  40
   18:  bipush  100
   20:  invokevirtual   #37; //Method Point.moveBy:(II)V
   23:  getstatic   #39; //Field java/lang/System.out:Ljava/io/
      PrintStream;
   26:  aload_0
   27:  invokevirtual   #16; //Method java/io/PrintStream.println:(Ljava/
      lang/Object;)V
   30:  return
```

Listagem 3.6: *Bytecode* do método doSomething costurado por *abc*.

```
   2:   iconst_5 //Segundo argumento
   3:   invokevirtual   #8; //Method Point.setPosition:(II)V
   6:   return
public static final void inline$1$around$0(int, int, Point, int, int);
  Code:
   0:   iload_0 //Carrega parâmetro x
   1:   istore  4
   3:   iload_1 //Carrega parâmetro y
   4:   istore_3
   5:   iload_0
   6:   ifge    12 //Se x > 0, pula para instr. 12
   9:   iconst_0 //Senão,
   10:  istore  4 //faz x = 0
   12:  iload_1
   13:  ifge    18 //Se y > 0, pula para instr. 18
   16:  iconst_0 //Senão,
   17:  istore_3 //faz y = 0
   18:  aload_2 //Carrega objeto-alvo
   19:  iload   4 //Carrega x
   21:  iload_3 //Carrega y
```

```
22:  invokevirtual   #8; //Method Point.setPosition:(II)V
25:  return
```

Listagem 3.7: Implementação especializada de um adendo de contorno para aplicação em doSomething

Observando o código da implementação do adendo, inlinecode1around$0, percebe-se que o problema de repetição de variáveis de contexto também aparece no código gerado pelo compilador *abc*. Esse problema é discutido na Seção 3.2.

3.1.2 Repetição de Adendos e Hachuras

Como descrito na Seção 2.2, 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 apresentado na Seção 3.1.1.

O combinador do compilador *ajc* insere implementações do adendo definido em CoordCheckAspect nas classes Point e Main. Cada hachura desse adendo é extraída para um método na classe onde ela aparece.

Note uma característica interessante deste programa: todas as hachuras do adendo de contorno existente são chamadas ao mesmo método, e diferem apenas pelo local da chamada e argumentos passados. Informações de contexto usadas na hachura, tais como argumentos, 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. O compilador *ajc* ignora essa repetição para todo adendo de contorno, embora o *abc* consiga detectá-la, durante a expansão, para adendos pequenos. Essa repetição, apesar de não parecer de grande impacto para um exemplo simples como o das listagens 3.1 e 3.2, pode replicar grandes porções de código em adendos de contorno complexos que se apliquem a vários pontos de junção em um mesmo programa.

Neste programa, o problema de implementações repetidas de adendos e hachuras aparece no código gerado para a classe Point, que tem duas hachuras de um mesmo adendo de contorno. Listagens 3.4 e 3.5 mostram o código gerado por *ajc* para implementar a aplicação do adendo no corpo do método doSomething da classe Main. Na classe Point, o combinador do compilador *ajc* gera dois pares de métodos idênticos a setPosition_aroundBody0 e setPosition_aroundBody1$advice. Um desses pares adendo-hachura

implementa a aplicação do adendo no corpo do construtor da classe Point, e o outro, no corpo do método moveBy (veja Listagens 3.1 e 3.2).

O problema de repetição de adendos e hachuras surge, portanto, quando uma classe c possui n hachuras, $h_1, h_2, \ldots, h_n$ de um adendo de contorno a, situação em que cada par (a, h_i) é introduzido nessa classe. A solução para esse problema consiste em eliminar os $n-1$ pares desnecessários do código gerado para a classe c e substituir referências, ou chamadas, a eles por referências ao seu correspondente remanescente, reduzindo assim o tamanho do código gerado para programas AspectJ que usem adendos de contorno.

Implementações repetidas de adendos e, em alguns casos, também de hachuras, são geradas pelo compilador *abc* durante a costura de adendos de contorno. A estratégia de expansão adotada na Versão 1.2.1 do compilador *abc* cria uma implementação do adendo para cada hachura na classe correspondente ao aspecto em que esse adendo foi definido. Entretanto, a hachura é expandida no corpo de implementações do adendo, de forma que, no código final gerado, não há métodos que implementem hachuras, ao contrário do *ajc*. Deve-se ressaltar que tais métodos não aparecem no código gerado, mas são, todavia, gerados durante a compilação em código intermediário, e expandidos após o término da costura de código.

Essa mesma estratégia de expansão permite ao *abc* detectar alguns casos de geração de métodos repetidos. Tal detecção depende da estrutura do código intermediário gerado durante a costura e, em especial, das variáveis locais capturadas no contexto de hachuras. Entretanto, a solução proposta neste trabalho permite a eliminação de implementações repetidas de adendos independentemente de tal estrutura, e é, portanto, mais eficaz do que a detecção realizada pelo *abc* durante a expansão de métodos.

3.2 Repetição de Variáveis de Contexto

A costura eficiente de adendos de contorno nos compiladores da linguagem AspectJ depende da expansão (*inlining*) do corpo de adendos, como mostrado na Seção 2.2.1. Hachuras afetadas por adendos de contorno são extraídas de seu local original e reposicionadas em métodos para a implementação do comando **proceed**. Para que a semântica da hachura seja mantida nesse processo, variáveis presentes em seu contexto, ou ambiente, original devem ser transferidas do local de aplicação do adendo de contorno para o local para onde a hachura foi movida.

Variáveis de contexto podem também ser capturadas pelo programador, por meio das cláusulas **this**, **target** e **args**, na expressão do conjunto de junção associada a um adendo. Como descrito na Seção 2.1, essas cláusulas disponibilizam no corpo do adendo o objeto atual em execução, o objeto alvo de chamadas, para conjuntos de junção do

tipo **call**, e os argumentos de um método respectivamente.

Não obstante, pode haver uma interseção entre as variáveis disponíveis no contexto de uma hachura extraída e as variáveis capturadas pelo programador no adendo aplicado a essa hachura. Os compiladores *ajc* e *abc* não detectam tal interseção, e capturam repetidamente algumas variáveis de contexto. A repetição de variáveis de contexto tem impacto no tamanho do código *bytecode* correspondente ao método expandido para implementação do adendo e também no tempo de execução de chamadas a esse método, uma vez que alguns argumentos são desnecessariamente carregados.

Considere o programa da Listagem 3.8, que consiste nas classes M e C. Uma representação textual do código produzido pelo compilador *ajc* para os métodos test1, test2 e test3 da classe M é mostrada na Listagem 3.9.

Como se pode observar no código da Listagem 3.9, o contexto disponível na chamada a m1, realizada por meio da instrução invokevirtual , contém apenas o objeto alvo dessa chamada, instância de C, recebido como argumento de test1 e carregado na pilha por meio da instrução aload_1. Em código *bytecode*, argumentos de métodos de instância são armazenados a partir da posição 1 de seu *frame*, e a instrução aload_1 carrega o valor nessa posição e o coloca no topo da pilha de execução. Os contextos dos métodos test2 e test3 contêm, além do objeto alvo das chamadas a m2 e m3, seus argumentos.

Suponha que o aspecto A da Listagem 3.10 seja aplicado a esse programa. Os adendos de contorno sobre os conjuntos de junção m1Calls e m2Calls não capturam contexto, e o adendo sobre m3Calls captura os argumentos passados ao método m3.

Listagem 3.11 mostra a implementação do adendo de contorno aplicado à chamada ao método m1 realizada em test1, na linha 9 da Listagem 3.8. O método m1_aroundBody1$advice é a expansão do adendo de contorno aplicado a essa chamada. A hachura desse adendo foi extraída para o método m1_aroundBody0, que recebe como parâmetros as variáveis de contexto necessárias para executá-la, e a chamada **proceed** na linha 6 da Listagem 3.10 é substituída, no corpo da expansão do adendo, por uma chamada a m1_aroundBody0. Como adendos podem acessar métodos e atributos dos aspectos em que são declarados, uma instância do aspecto é passada para toda expansão de adendo.

O método m2 recebe como argumentos dois valores inteiros. Listagem 3.12 mostra o código resultante da aplicação do adendo de contorno associado ao conjunto de junção m2Calls à chamada a m2 realizada na linha 10 da Listagem 3.8. Além da instância de C necessária para a chamada a m2, a expansão do adendo e a própria hachura recebem também os argumentos dessa chamada.

O adendo de contorno definido nas linhas 11 a 13 da Listagem 3.10 captura os argumentos de chamadas a m3. Os argumentos da chamada a esse método estão disponíveis tanto no ambiente da hachura quanto no contexto capturado pela expressão

```
1  public class M {
2      public static void main(String[] args) { new M().run(); }
3      public void run() {
4          C c = new C();
5          test1(c);
6          test2(c);
7          test3(c);
8      }
9      private void test1(C c) { c.m1(); }
10     private void test2(C c) { c.m2(27, 28); }
11     private void test3(C c) { c.m3(14, 57); }
12 }
13 public class C {
14     public void m1() {
15         System.out.println("m1()");
16     }
17     public void m2(int x, int y) {
18         System.out.println("m2(" + x + "," + y + ")");
19     }
20     public void m3(int x, int y) {
21         System.out.println("m3(" + x + "," + y + ")");
22     }
23 }
```

Listagem 3.8: Programa simples em Java.

```
private void test1(C);
  Code:
    0:   aload_1
    1:   invokevirtual   #47; //Method C.m1:()V
    4:   return
private void test2(C);
  Code:
    0:   aload_1
    1:   bipush  27
    3:   bipush  28
    5:   invokevirtual   #44; //Method C.m2:(II)V
    8:   return
private void test3(C);
  Code:
    0:   aload_1
    1:   bipush  14
    3:   bipush  57
    5:   invokevirtual   #41; //Method C.m3:(II)V
    8:   return
```

Listagem 3.9: Representação textual do código *bytecode* dos métodos test1, test2 e test3 da classe M.

```
1  public aspect A {
2      pointcut m1Calls(): call(void C.m1());
3      pointcut m2Calls(): call(void C.m2(int,int));
4      pointcut m3Calls(int x, int y): call(void C.m3(int,int)) && args(x,y);
5      void around(): m1Calls() {
6          proceed();
7      }
8      void around(): m2Calls() {
9          proceed();
10     }
11     void around(int x, int y): m3Calls(x,y) {
12         proceed(x,y);
13     }
14 }
```

Listagem 3.10: Aspecto que captura contexto em chamadas aos métodos de C.

```
private void test1(C);
  Code:
    0:   aload_1
    1:   astore_2
    2:   aload_0 //Carrega objeto this
    3:   aload_2 //Carrega parâmetro c
    4:   invokestatic    #62; //Method A.aspectOf:()LA;
    7:   aconst_null //Não há closure
    8:   invokestatic    #66; //Method m1_aroundBody1$advice:(LM;LC;LA;
       Lorg/aspectj/runtime/internal/AroundClosure;)V
    11:  return
private static final void m1_aroundBody0(M, C);
  Code:
    0:   aload_1 //Carrega objeto-alvo c
    1:   invokevirtual   #40; //Method C.m1:()V
    4:   return
private static final void m1_aroundBody1$advice(M, C, A, org.aspectj.
    runtime.internal.AroundClosure);
  Code:
    0:   aload_3
    1:   astore  4
    3:   aload_0 //Carrega instância de M
    4:   aload_1 //Carrega instância de C
    5:   invokestatic    #68; //Method m1_aroundBody0:(LM;LC;)V
    8:   return
```

Listagem 3.11: Implementação do adendo de contorno aplicado ao corpo de test1.

```
private void test2(C);
  Code:
    0:   aload_1
    1:   bipush  27
    3:   bipush  28
    5:   istore_2
    6:   istore_3
    7:   astore  4
    9:   aload_0 //Carrega objeto this
    10:  aload   4 //Carrega parâmetro c
    12:  iload_3 //Carrega constante 27
    13:  iload_2 //Carrega constante 28
    14:  invokestatic    #62; //Method A.aspectOf:()LA;
    17:  aconst_null //Não há closure
    18:  invokestatic    #77; //Method m2_aroundBody3$advice:(LM;LC;IILA;
       Lorg/aspectj/runtime/internal/AroundClosure;)V
    21:  return
private static final void m2_aroundBody2(M, C, int, int);
  Code:
    0:   aload_1
    1:   iload_2
    2:   iload_3
    3:   invokevirtual   #44; //Method C.m2:(II)V
    6:   return
private static final void m2_aroundBody3$advice(M, C, int, int, A, org.
    aspectj.runtime.internal.AroundClosure);
  Code:
    0:   aload   5
    2:   astore  6
    4:   aload_0 //Carrega instância de M
    5:   aload_1 //Carrega instância de C
    6:   iload_2 //Primeiro argumento
    7:   iload_3 //Segundo argumento
    8:   invokestatic    #79; //Method m2_aroundBody2:(LM;LC;II)V
    11:  return
```

Listagem 3.12: Implementação do adendo de contorno aplicado ao corpo de test2.

de conjunto de junção m3Calls. No *bytecode* criado para implementar essa aplicação de adendo, mostrado na Listagem 3.13, o método m3_aroundBody5$advice recebe cada uma dessas variáveis duas vezes: a primeira resulta da captura de contexto necessária para mover a hachura de seu local original para o método m3_aroundBody4, e a segunda decorre da captura explícita realizada pelo programador em m3Calls.

Variáveis de contexto repetidas são desnecessárias, já que recebem os mesmos argumentos, provenientes do local de onde a hachura é extraída. O *bytecode* da expansão do adendo de contorno reflete essa característica: os valores disponíveis nas posições 2 e 3 do *frame* do método m3_aroundBody5$advice, correspondentes ao seu primeiro par de

parâmetros inteiros, não é usado no corpo do método. É possível, portanto, eliminar variáveis de contexto repetidas do *bytecode* gerado na costura de adendos de contorno realizada pelo compilador *ajc* sem modificar sua semântica.

```
private void test3(C);
  Code:
    0:   aload_1
    1:   bipush  14
    3:   bipush  57
    5:   istore_2
    6:   istore_3
    7:   astore  4
    9:   aload_0 //Carrega objeto this
    10:  aload   4 //Carrega parâmetro c
    12:  iload_3 //Carrega constante 14
    13:  iload_2 //Carrega constante 57
    14:  invokestatic    #62; //Method A.aspectOf:()LA;
    17:  iload_3 //Carrega constante 14
    18:  iload_2 //Carrega constante 57
    19:  aconst_null //Não há closure
    20:  invokestatic    #88; //Method m3_aroundBody5$advice:(LM;LC;IILA;
       IILorg/aspectj/runtime/internal/AroundClosure;)V
    23:  return
private static final void m3_aroundBody4(M, C, int, int);
  Code:
    0:   aload_1
    1:   iload_2
    2:   iload_3
    3:   invokevirtual   #47; //Method C.m3:(II)V
    6:   return
private static final void m3_aroundBody5$advice(M, C, int, int, A, int,
    int, org.aspectj.runtime.internal.AroundClosure);
  Code:
    0:   iload   5
    2:   iload   6
    4:   aload   7
    6:   astore  8
    8:   istore  9
    10:  istore  10
    12:  aload_0 //Carrega instância de M
    13:  aload_1 //Carrega instância de C
    14:  iload   10 //Primeiro argumento
    16:  iload   9 //Segundo argumento
    18:  invokestatic    #90; //Method m3_aroundBody4:(LM;LC;II)V
    21:  return
```

Listagem 3.13: Implementação do adendo de contorno aplicado ao corpo de test3.

Código gerado pelo compilador *abc* para a costura de adendos de contorno também

apresenta variáveis de contexto repetidas[3]. Sua estratégia de passagem de contexto para hachuras separa as etapas de identificação de contexto implicitamente necessário para a execução de hachuras e contexto explícitamente capturado pelo programador, assim como no *ajc*. A ligação entre variáveis locais explicitamente capturadas pelo programador e parâmetros de implementações de adendos de contorno é realizada pelo *front-end* do *abc*, e a determinação de variáveis necessárias para a execução da hachura é dada durante a costura de código.

3.3 Conclusões

Este capítulo apresentou, por meio da análise de código gerado pelos compiladores *ajc* e *abc*, os problemas de repetição de implementações de adendos e hachuras, e de repetição de variáveis de contexto.

Implementações repetidas de adendos e hachuras são introduzidas na costura de adendos de contorno quando várias hachuras idênticas de um mesmo adendo aparecem em uma classe, e são resultantes da falha desses compiladores em detectar que a semântica de um mesmo adendo de contorno aplicado a diferentes hachuras é a mesma para todas essas hachuras, salvo elementos como argumentos e variáveis locais que, entretanto, fazem parte do contexto passado para esses adendos. Esse problema causa um aumento desnecessário no tamanho do código de programas AspectJ.

Variáveis de contexto repetidas também aparecem em código gerado durante a costura de adendos de contorno. São resultantes da captura separada de variáveis locais necessárias para a correta execução de hachuras e das variáveis explicitamente capturadas pelo programador em pontos de junção. Esse problema causa aumentos desnecessários no tamanho, no tempo de execução e no consumo de memória de programas AspectJ.

O restante deste texto apresenta soluções para os problemas discutidos neste capítulo, e analisa os resultados obtidos a partir da implementação dessas soluções.

[3] Optou-se por não apresentar código gerado pelo compilador *abc* com esse problema, por ser similar ao gerado pelo *ajc*.

Capítulo 4

União de Pontos de Junção

Compiladores da linguagem AspectJ introduzem, no código *bytecode* de programas Java, comportamentos transversais definidos por aspectos. O código de adendos de contorno introduzido em um programa, durante o processo de costura dos compiladores *ajc* e *abc*, contém métodos *equivalentes*. Métodos equivalentes são aqueles que possuem a mesma lista de instruções, os mesmos argumentos e o mesmo tipo de retorno. Uma definição mais cuidadosa é dada na Seção 4.1.2. Por serem equivalentes a outros, alguns desses métodos são desnecessários, e sua remoção diminui o tamanho do código *bytecode* do programa sem modificar sua semântica.

Esta seção apresenta soluções para o problema de repetição de código existente nos compiladores *ajc* e *abc*. Há duas abordagens de implementação dessa técnica. Na primeira, pós-compilação, o código *bytecode* gerado pelos compiladores é analisado em uma etapa separada à compilação, e os métodos desnecessários gerados na costura de adendos são *removidos* do código. Essa primeira abordagem serve como uma prova de conceito da técnica aqui proposta. A segunda abordagem, de cunho prático, tem por objetivo *evitar a criação* de réplicas de métodos durante a costura de código, e é integrada aos compiladores *ajc* e *abc* para que se realize durante a compilação.

4.1 União Pós-Compilação

Implementar a união de pontos de junção como um passo posterior à compilação permite avaliar e corrigir, caso necessário, a solução proposta, antes ainda de estudar a estratégia de integração dessa solução aos compiladores *ajc* e *abc*. Além disso, a implementação inicial dessa estratégia permite avaliar as tecnologias usadas no trabalho, e verificar a aplicabilidade de algumas ferramentas disponíveis para manipulação de *bytecode*. Esta seção descreve o desenvolvimento de um programa que elimina réplicas de métodos de implementação de adendos e hachuras geradas pelos compiladores *ajc* e

abc.

O algoritmo pós-compilação difere da solução final principalmente nas estruturas disponíveis para identificação das réplicas de métodos de implementação de adendos de contorno. Na etapa de casamento (*matching*), compiladores da linguagem AspectJ geram, a partir do código-fonte, dados que informam quais pontos do programa são hachuras de adendos, e guardam essas informações para as fases posteriores, como geração de código e costura de adendos. Após a compilação, tais informações não estão mais disponíveis, de forma que o algoritmo de união que executa nesse estágio deve se basear apenas no código gerado pelo compilador para identificar os métodos sobre os quais deve operar. Entretanto, métodos gerados na costura de adendos possuem nomes gerados automaticamente, de forma que é simples identificá-los sistematicamente no código *bytecode* produzido pelos compiladores *ajc* e *abc*.

4.1.1 Ferramentas

Compiladores da linguagem AspectJ realizam costura de código sobre *bytecode*, e não código Java. *Bytecode* é um formato de código de máquina baseado em uma pilha de execução, no qual instruções e seus argumentos são representados em um ou mais *bytes* [LY99]. Apenas um *byte* é usado para identificar uma dentre as 255 instruções *bytecode*. A interpretação desse código é uma tarefa trabalhosa, porém simplificada por ferramentas de representação e manipulação de *bytecode*.

No desenvolvimento do presente trabalho, três ferramentas de manipulação de *bytecode* foram estudadas: BCEL, Soot[1] e ASM. Esta seção descreve as vantagens e desvantagens de cada uma e descreve os motivos que levaram à escolha da biblioteca ASM para a implementação do algoritmo de união de pontos de junção.

BCEL

A biblioteca *ByteCode Engineering Library* (BCEL) [DvZH03] define uma representação de um arquivo *class*, ou seja, um arquivo que define uma classe Java em *bytecode*. Em BCEL, existe um repositório de classes, de onde são obtidas representações de classes de um programa para análise e manipulação. Uma versão modificada dessa biblioteca é usada no compilador oficial da linguagem AspectJ, *ajc*.

Cada classe carregada em BCEL é representada por um objeto JavaClass, que fornece acesso aos seus atributos, métodos e outras informações. É possível também escrever programas, usando BCEL, que geram novas classes por meio da classe ClassGen. Classes criadas ou modificadas pelo BCEL podem ser carregadas em tempo de execução, por meio de carregadores de classes (*class loaders*), ou gravadas em arquivo.

[1] Soot não é um arcabouço de representação de *bytecode*, mas sim de otimização e transformação

Embora BCEL seja uma ferramenta poderosa para manipulação de *bytecode*, a representação que ele provê desse código é muito próxima do formato de arquivos *class*, incluindo detalhes como o repositório de constantes de uma classe (*Constant Pool*), de onde classes e referências para métodos e atributos devem ser recuperados. Dessa forma, programas que utilizam BCEL manipulam em baixo nível a estrutura de arquivos *class*, e não apenas as classes que eles descrevem.

O desenvolvimento dessa biblioteca foi interrompido em abril de 2003, e, desde então, apenas pequenas correções de erros foram lançadas. Suporte a Java 5 foi desenvolvido como parte dos desenvolvedores do compilador *ajc*, mas essa modificação não foi incluída na distribuição oficial de BCEL.

O baixo nível de abstração da biblioteca BCEL e a descontinuidade do seu projeto levaram à busca, no desenvolvimento deste trabalho, por ferramentas mais atuais e que ofereçam uma maior abstração sobre *bytecode*.

Soot

Soot é um arcabouço de otimização de código Java que fornece diversas representações intermediárias para análise e transformação de *bytecode*[VGH+00, VHS+99]. Esse arcabouço define quatro linguagens intermediárias para manipulação de *bytecode*, chamadas *Baf*, *Jimple*, *Shimple* e *Grimp*. Ao contrário de BCEL, o uso de representações de mais alto nível de *bytecode*, e em especial a representação Jimple, abstraem diversos detalhes de arquivos *class* irrelevantes para este trabalho. Soot é o arcabouço utilizado para geração e otimização de código no compilador *abc*.

A representação Jimple usada no Soot consiste em um código de três endereços tipado, que abstrai a pilha de execução do *bytecode*. Para a implementação de otimizações, Soot ainda provê todas as classes no *classpath* de um programa; a tarefa de encontrar essas classes é delegada ao usuário das bibliotecas BCEL e ASM.

Muito embora Soot contenha uma representação de classes, seus métodos e atributos, esse não é o seu objetivo. Como um arcabouço de otimização de código Java, Soot define um conjunto pré-definido de otimizações e uma estrutura para inclusão de novas análises e transformações no seu processo de otimização. Apesar de a inclusão de um novo transformador no processo de otimização do Soot ser simples, ela implica em todas as otimizações do arcabouço serem executadas, o que se torna um problema dessa arquitetura, pois algumas operações desse processo possuem alto custo de execução. Por exemplo, para montar as informações necessárias para executar otimizações que necessitem de análise de todo o programa, por exemplo, Soot cria de um grafo de chamadas de métodos, cuja complexidade é $O(n^2)$, onde n é o número de métodos do

programa[2].

ASM

O baixo nível de abstração da biblioteca BCEL e o desalinhamento entre os requisitos deste trabalho e as funcionalidades providas pelo arcabouço Soot levaram à busca por outra ferramenta para interpretação e manipulação de *bytecode*.

A biblioteca ASM [Kul05] oferece abstração de detalhes do *bytecode* permitindo, todavia, transformar código existente ou gerar código. Ela é eficiente pois evita construir uma representação de *bytecode* em memória: ASM lê um arquivo *class* e lança eventos ao identificar estruturas como classes, métodos e atributos. Esses eventos são tratados por implementações do padrão de projeto *Visitor* [GHJV95], que podem remover, modificar ou deixar inalterados os elementos encontrados.

Para casos em que a representação em memória de uma classe é necessária, pode-se usar o pacote *asm-tree*, distribuído separadamente. Esse pacote implementa um *visitor* que, ao encontrar estruturas em arquivos *class*, cria objetos correspondentes em uma representação de árvore de *bytecode*, de forma similar à biblioteca BCEL.

Embora a metodologia de implementar *visitors* para modificar e remover elementos de código *bytecode* seja, a princípio, pouco intuitiva, código escrito na biblioteca ASM é, em geral, simples, elegante e pequeno. A escolha de ASM ajusta-se perfeitamente na implementação do algoritmo proposto nesta seção, já que o uso de sua estratégia baseada em eventos de manipulação de *bytecode* pode ser confinado aos módulos em que a biblioteca é utilizada. Além disso, como nenhuma representação de *bytecode* é usada em ASM, o restante do programa não precisa lidar com tais estruturas, o que mantém a estrutura geral do algoritmo independente da ferramenta usada na implementação.

4.1.2 Algoritmo Proposto

O objetivo principal do algoritmo descrito nesta seção, chamado de "União de Pontos de Junção e Hachuras", é eliminar repetições de métodos gerados na costura de adendos de contorno pelos compiladores *abc* e *ajc*. Diversos objetivos secundários surgem como passos desse algoritmo, tais como identificação de todas as classes de um programa e adaptação do algoritmo para especificidades de código gerado por diferentes compiladores. A seqüência de passos do algoritmo é:

1. identificar conjunto C de classes da aplicação;

2. para cada classe $c \in C$:

[2] O pior caso é aquele em que cada método do programa chama todos os demais, inclusive ele próprio.

2.1. identificar conjunto T de métodos-alvo;

2.2. montar as classes de equivalência de métodos-alvo;

2.3. escolher os "pivôs"[3] das classes de equivalência;

2.4. remover todos os métodos-alvo da classe Java c, exceto os pivôs;

3. substituir chamadas a métodos removidos por chamadas aos seus pivôs;

4. gravar as classes resultantes.

Métodos-alvo são aqueles gerados pelos compiladores *ajc* e *abc* para implementar adendos e hachuras. Como esse algoritmo é independente da compilação, métodos-alvo devem ser identificados diretamente a partir do *bytecode*. Na implementação descrita na Seção 4.1.2, esses métodos são identificados por meio do casamento dos nomes dos métodos a expressões regulares que descrevem métodos gerados para implementar adendos e hachuras.

Nessa descrição em alto nível do algoritmo, afirma-se que classes de equivalência são montadas a partir de métodos-alvo. A relação que define tais classes de equivalência é descrita na Definição 1.

Definição 1. Sejam c uma classe Java e E uma relação sobre o universo M de métodos da classe c. Dois métodos m e n pertencentes a c são equivalentes, denotado por $(m, n) \in E$, se, e somente se, as seguintes condições forem satisfeitas:

(i) A assinatura de m é igual à de n, ou seja, as listas de parâmetros de m e n possuem mesmos tipos e na mesma ordem, e o tipo de retorno dos dois métodos é o mesmo;

(ii) As listas de instruções de m e n possuem o mesmo tamanho e são iguais para-a-par, ou seja, as mesmas instruções aparecem em m e n na mesma ordem e, quando houver, com os mesmos argumentos.

□

O Teorema 1 mostra que a relação da Definição 1 é, de fato, uma relação de equivalência.

Teorema 1. *A relação E da Definição 1 particiona o universo dos métodos de uma classe Java em classes de equivalência, ou seja, a relação E é reflexiva, simétrica e transitiva.*

[3] Pivô de uma classe de equivalência é um elemento qualquer dessa classe, usado para representar os demais. Ele é equivalente, de acordo com a relação E da Definição 1, a todos os outros elementos de sua classe de equivalência.

Prova *demonstra-se separadaramente que a relação E é reflexiva, simétrica e transitiva, e, portanto, é uma relação de equivalência sobre o universo dos métodos de uma classe Java. Seja um método Java m definido como uma tupla (A_m, r_m, I_m), onde A_m é a assinatura de m, ou seja, sua lista de parâmetros, r_m é o seu tipo de retorno e I_m seu corpo ou lista de instruções.*

E **é reflexiva.** *claramente, para todo método de uma classe Java, sua lista de argumentos, seu tipo de retorno e sua lista de instruções são iguais a si próprios e, portanto, pela Definição 1, E é reflexiva.*

E **é simétrica.** *suponha que $m = (A_m, r_m, I_m)$, $n = (A_n, r_n, I_n)$ e $(m, n) \in E$. Então, pela Definição 1, $A_m = A_n$, $r_m = r_n$ e $I_m = I_n$; mas então claramente $A_n = A_m$, $r_n = r_m$ e $I_n = I_m$ e, assim, $(n, m) \in E$. Portanto, como m e n são arbitrários, a relação E é simétrica.*

E **é transitiva.** *suponha que $m = (A_m, r_m, I_m)$, $n = (A_n, r_n, I_n)$ e $o = (A_o, r_o, I_o)$ são métodos de uma classe c, $(m, n) \in E$ e $(n, o) \in E$. De $(m, n) \in E$ segue que $m = (A_n, r_n, I_n)$. Mas então de $(n, o) \in E$ conclui-se que $((A_n, r_n, I_n), o) \in E$ e, assim, $(m, o) \in E$. Portanto, como m, n e o são arbitrários, a relação E é transitiva.*

As classes de equivalência em que a relação E da Definição 1 particiona o universo dos métodos de uma classe Java c, no contexto do algoritmo em presente descrição, são chamadas também de *conjuntos de réplicas*.

Uma vez montados conjuntos de réplicas de métodos de implementação de adendos e hachuras, basta eliminar da classe c todos, exceto um, desses métodos. O método restante de uma classe de equivalência é chamado de "pivô". Nesse ponto da execução do algoritmo, o programa em transformação torna-se inconsistente: é possível que existam chamadas a métodos removidos. Segue-se então a substituição de chamadas a métodos removidos por chamadas aos seus pivôs. Tal substituição não modifica a semântica do programa, já que, por construção, cada método removido m de uma classe de equivalência R é equivalente ao seu pivô p, onde R é uma das classes de equivalências produzidas pela relação de equivalência da Definição 1. Dessa forma, qualquer chamada a p obtém, dada uma mesma lista de argumentos, o mesmo resultado de uma chamada a m.

O restante desta seção descreve o algoritmo proposto para união de implementações de adendos e hachuras. Nos pseudo-códigos que aparecem a seguir, segue-se a convenção de que procedimentos cujos nomes são sublinhados, como PIVOT e IS-METHOD-CALL, são pontos de extensão, isto é, dependentes de especificidades de implementação. A semântica esperada de cada procedimento é descrita em seu primeiro uso.

```
Advice-Shadow-Union(main-class)
1  C ← Find-App-Classes(main-class)
2  R ← ∅
3  for each c ∈ C
4        do T ← Identify-Targets(c)
5           R[c] ← Find-Replicas(T)
6           Remove-Methods(c)
7  for each c ∈ C
8        do Replace-Calls(c, R)
```

O procedimento Advice-Shadow-Union implementa a seqüência de passos descrita no início desta seção. Inicialmente, busca-se, por meio do procedimento Find-App-Classes, o conjunto C de classes do programa. Cria-se, então, uma família de relações de equivalência R, vazia; durante o algoritmo, cada elemento $R[c]$ de R representa a relação de equivalência para métodos da classe c, onde $c \in C$.

Para cada classe c, nas linhas 4 a 6 de Advice-Shadow-Union, constroem-se o conjunto de métodos-alvo T da classe e a relação de equivalência entre eles, $R[c]$, e removem-se todos os métodos, exceto o pivô, dos conjuntos de réplicas em $R[c]$. Após todas as classes terem sido analisadas e todas as réplicas de métodos removidas, percorre-se novamente as classes da aplicação, nas linhas 7 e 8, substituindo chamadas aos métodos removidos na linha 6 por chamadas aos seus pivôs.

O procedimento Find-App-Classes encontra o conjunto de classes referenciadas em um programa, a partir da classe passada para ele como argumento. O procedimento Navigate é utilizado para construir, recursivamente, o conjunto de classes da aplicação.

```
Find-App-Classes(main-class)
1  C ← ∅
2  C ← Navigate(main-class, C)
3  return C
```

```
Navigate(c, C)
1  C ← C ∪ {c}
2  for each c' ∈ ref[c]
3        do if c' ∉ C
4              then C ← C ∪ Navigate(c', C)
5  return C
```

A expressão $ref[c]$, usada na linha 2 de Navigate, denota o conjunto de classes *referenciadas* na classe c. No contexto de programas Java, diz-se que $c' \in ref[c]$ se, e somente se:

- c' é o tipo de algum atributo de c; ou

- c' é o tipo da superclasse de c; ou
- c' é uma interface implementada por c; ou
- c' é tipo de um parâmetro ou tipo de retorno de algum método de c; ou
- c' é um tipo usado no corpo de algum método de c; ou
- c' é uma classe interna de c; ou
- existe alguma classe c'' tal que c'' é uma classe interna de c e $c' \in ref[c'']$.

O procedimento IDENTIFY-TARGETS constrói e retorna o conjunto T de métodos-alvo para uma dada classe c. A variável $methods[c]$ usada na linha 2 desse procedimento denota o conjunto de métodos da classe c. IDENTIFY-TARGETS verifica, então, por meio do procedimento IS-TARGET, quais métodos são passíveis de eliminação, ou seja, quais podem ser considerados para verificação de existência de réplicas. IS-TARGET é um ponto de extensão pois depende do ambiente em que o algoritmo é executado. Há várias maneiras pelas quais alvos podem ser identificados. Como descrito na Seção 4.1.3, na implementação atual do algoritmo, alvos são identificados por meio do casamento de padrões, definidos por meio de expressões regulares, aos seus nomes.

```
IDENTIFY-TARGETS(c)
1  T ← ∅
2  for each m ∈ methods[c]
3        do if IS-TARGET(m)
4              then T ← T ∪ {m}
5  return T
```

Conhecido o conjunto T de métodos-alvo do algoritmo, o procedimento FIND-REPLICAS é responsável por construir uma relação de equivalência a partir desses alvos, de acordo com a Definição 1. O método ADD-TO-EQUIV-CLASS, usado na linha 3 do procedimento FIND-REPLICAS, é responsável por encontrar, em R, a classe de equivalência adequada para um dado método-alvo t. A família de conjuntos de réplicas construída é retornada no fim.

```
FIND-REPLICAS(T)
1  R ← ∅
2  for each t ∈ T
3        do ADD-TO-EQUIV-CLASS(t, R)
4  return R
```

Para cada conjunto de réplicas, *replica-set*, de uma classe c, o procedimento Remove-Methods escolhe, na linha 2, um pivô p. As réplicas de p, denotadas por *replica-set* $-\{p\}$, são removidas na linha 3. O procedimento Pivot-For-Equiv-Class é um ponto de extensão pois depende da implementação de relação de equivalência usada; seu resultado é o pivô da classe de equivalência passada como argumento – para todas as chamadas de Pivot-For-Equiv-Class com uma mesma classe de equivalência, o pivô escolhido é o mesmo.

```
Remove-Methods(c)
1  for each replica-set ∈ R[c]
2       do p ← Pivot-For-Equiv-Class(replica-set)
3          methods[c] ← (methods[c] − replica-set[c]) ∪ {p}
```

Voltando ao algoritmo Advice-Shadow-Union, na Página 75, o último passo da eliminação de réplicas é garantir a consistência do programa modificado, ou seja, substituir chamadas a métodos removidos na linha 6 por chamadas aos pivôs de suas classes de equivalência. O procedimento Replace-Calls é chamado, então, para cada classe c do programa. Esse procedimento percorre, então, todas as instruções de cada método m de c; a lista de instruções de um método m é denotada por *instructions*$[m]$. Se uma instrução i for uma chamada de método, verificação realizada pela chamada ao procedimento externo Is-Method-Call, o método chamado por i, *called-method*$[i]$, é então substituído pelo pivô desse método, Pivot-For-Element(*called-method*$[i], R$).

Verificar se uma instrução é chamada de método, operação realizada pelo procedimento Is-Method-Call, depende da ferramenta de manipulação de *bytecode* usada na implementação e, portanto, essa operação é considerada um ponto de extensão do algoritmo aqui apresentado; as instruções de chamada de método em *bytecode* são `invokevirtual`, `invokeinterface`, `invokespecial` e `invokestatic`.

O procedimento Pivot-For-Element usado em Replace-Calls difere levemente daquele usado em Remove-Methods, Pivot-For-Equiv-Class, porque, em vez de escolher o pivô de uma dada classe de equivalência, ele escolhe o pivô de um elemento da relação de equivalência. Essa tarefa consiste em encontrar a classe de equivalência do elemento (método) passado como argumento e, então, escolher o pivô dessa classe.

```
Replace-Calls(c, R)
1  for each m ∈ methods[c]
2       do for each i ∈ instructions[m]
3             do if Is-Method-Call(i)
4                then called-method[i] ← Pivot-For-Element(called-method[i], R)
```

Análise de Complexidade

A complexidade do algoritmo proposto nesta seção depende da complexidade de seus pontos de extensão, ou seja, da complexidade de operações externas, dependentes de decisões de implementação, que são usadas em alguns passos, como IS-TARGET e IS-METHOD-CALL. Considera-se, então, na corrente análise, que tais operações possuem complexidade $O(1)$ – uma implementação que atende a esse requisito é descrita na Seção 4.1.3.

Sejam $m[c]$ o número de métodos de uma classe Java c, $i[c]$ o número de instruções da classe c, $I[P]$ o número total de instruções de um programa P, $\alpha[c]$ o número de aplicações de adendos de contorno na classe c e $\alpha[P]$ o número total de aplicações de adendos de contorno. O algoritmo FIND-APP-CLASSES, que encontra o conjunto C de classes de um programa a partir da classe que inicia sua execução, tem complexidade $O(I[P])$, pois busca recursivamente, nas instruções do programa P novas classes referenciadas a partir do conjunto C calculado até o momento. Na implementação apresentada na Seção 4.1.3, esse algoritmo é implementado como uma busca em amplitude pelas classes do programa.

O algoritmo IDENTIFY-TARGETS testa todos os métodos do programa P. A complexidade desse algoritmo, considerando-se que o algoritmo IS-TARGET tem tempo de execução constante, é $O(m[c])$, ou seja, IDENTIFY-TARGETS é linear em relação ao número de métodos da classe c que recebe como argumento. O algoritmo FIND-REPLICAS não opera sobre todos os métodos de uma classe c, mas apenas sobre aqueles métodos identificados como alvos em IDENTIFY-TARGETS, ou seja, métodos que implementam adendos e hachuras. Sua complexidade é, portanto, linear em relação ao número de aplicações de adendos de contorno que possuem hachuras na classe c, ou seja, $O(\alpha[c])$, pois o algoritmo ADD-TO-EQUIV-CLASS tem tempo de execução constante.

Embora a complexidade do algoritmo ADD-TO-EQUIV-CLASS seja considerada $O(1)$, essa complexidade é verdadeira apenas no *caso médio*. Uma classe de equivalência pode ser implementada eficientemente como uma tabela *hash* que mapeia métodos (pivôs) a listas de suas réplicas. Com uma boa função *hash*, a inclusão de um método m em uma classe de equivalência causará apenas uma comparação instrução por instrução entre m e o seu pivô. Em outras palavras, para uma boa função *hash* h sobre métodos m e n, a probabilidade do caso em que $h(m) = h(n)$ e $(m, n) \notin E$, onde E é a relação de equivalência da Definição 1, é próxima de zero. Assim, a inserção de um novo método t em uma partição R, na linha 3 do algoritmo FIND-REPLICAS, executa na média em tempo $O(1)$, realizando apenas uma comparação instrução por instrução entre esse método e seu pivô. Essa abordagem foi adotada na implementação da Seção 4.1.3, e a função $h(m)$ usada nessa implementação produz um valor inteiro a

partir da assinatura e do número de instruções de m.

A remoção de métodos desnecessários de uma classe c realizada pelo algoritmo Remove-Methods tem complexidade $O(\alpha[c])$. Nesse algoritmo, os pivôs de todas as classes de equivalência de métodos da classe c são obtidos, e suas réplicas são removidas do conjunto $methods[c]$. O algoritmo Replace-Calls, que substitui as chamadas a esses métodos, percorre todas as instruções de uma classe do programa para encontrar tais chamadas, e, portanto, possui complexidade $O(i[c])$ – note que o pivô de um método pode ser obtido em tempo $O(1)$ a partir do algoritmo Pivot-For-Element.

No algoritmo Advice-Shadow-Union (pág. 75), os algoritmos Identify-Targets, Find-Replicas, Remove-Methods e Replace-Calls são executados para todas as classes do programa. Seja $T_q(i)$ o tempo de execução do algoritmo q para uma entrada i. A complexidade de Advice-Shadow-Union para um programa $P = (C, A)$, onde C é o conjunto de classes do programa P e A é o conjunto de adendos de contorno, é:

$$\begin{aligned} T_{\text{Advice-Shadow-Union}}(P) &= T_{\text{Find-App-Classes}}(C[P]) + \\ &\quad \sum_{c \in C} [T_{\text{Identify-Targets}}(c) + T_{\text{Find-Replicas}}(t) \\ &\quad + T_{\text{Remove-Methods}}(c) + T_{\text{Replace-Calls}}(c)] \\ &= O(I[P] + M[P] + \alpha[P] + \alpha[P] + I[P]) \\ &= O(I[P]) \end{aligned}$$

Como o número de métodos e aplicações de adendos de um programa é menor do que o seu número de instruções, pois cada método e cada aplicação de adendo possui pelo menos uma instrução, o algoritmo Advice-Shadow-Union é linear em relação ao número de instruções do programa de entrada.

4.1.3 Implementação

O algoritmo descrito na Seção 4.1.2 foi implementado em um programa Java que elimina réplicas de métodos gerados pelos compiladores *ajc* e *abc*. A classe principal de um programa, cujo código tenha sido gerado por um desses compiladores, é um dos argumentos da implementação proposta. A implementação recebe ainda, como argumento, a especificação de qual compilador gerou o código a ser transformado para adaptar a sua definição do método correspondente ao procedimento Is-Target usado no algoritmo.

A definição de Is-Target fornecida por essa implementação utiliza expressões regulares para identificar os métodos gerados pelos compiladores *abc* e *ajc* durante a

costura de adendos de contorno. Para que um único programa seja capaz de lidar com *bytecode* gerado pelos dois compiladores, sem espalhamento das especificidades de cada um por todo o código, utiliza-se uma fábrica de configurações que, dada a identificação do compilador que gerou o código de entrada, cria uma "configuração de especialização". A configuração é responsável por determinar as expressões regulares que determinam o padrão de nomenclatura de uma implementação de adendos (adviceMethodPattern) e de hachuras (shadowMethodPattern).

Especializações já implementadas são capazes de identificar métodos gerados pelos compiladores *abc*, Versão 1.2.0, e *ajc*, Versão 5. Os padrões de nomes para implementações de adendos e hachuras desses dois compiladores são mostrados na Tabela 4.1. Nessa tabela, ocorrências de $\mathcal{L}$ e $\mathcal{D}$ denotam, respectivamente, uma letra e um dígito.

Compilador	Implementações de adendo	Implementações de hachura
ajc 5	$\mathcal{L}(\mathcal{L}\|\mathcal{D}\|\_)*$″ $\_aroundBody″\mathcal{D}+$ ″$advice″	$\mathcal{L}(\mathcal{L}\|\mathcal{D}\|\_)*$″ $\_aroundBody″\mathcal{D}+$
abc 1.2.0	″inline$″\mathcal{D}+$″ $around$″\mathcal{D}+$	″shadow$″\mathcal{D}+$

Tabela 4.1: Padrões de nomes de métodos gerados na costura de adendos de contorno

Figura 4.1 mostra um diagrama de classes com as principais classes da implementação apresentada nesta seção. A classe ShadowUnion implementa a seqüência de passos do algoritmo ADVICE-SHADOW-UNION. Os procedimentos auxiliares utilizados no algoritmo principal, exceto por FIND-REPLICAS, foram implementados em classes separadas, já que dependem diretamente da biblioteca ASM para interpretração e manipulação de *bytecode*; dessa forma, caso haja necessidade de utilizar outra biblioteca, basta substituir as implementações dessas classes.

O mesmo algoritmo é aplicado duas vezes sobre o programa de entrada, para eliminar réplicas de hachuras em um primeiro passo, e eliminar réplicas de adendos no segundo. Os métodos-alvo da primeira execução do algoritmo são implementações de hachuras. Após a remoção de réplicas de hachuras, removem-se réplicas de adendos. Suponha que duas implementações de adendo a_1 e a_2 sejam geradas na costura de um adendo de contorno a na classe c. Em a_1, a chamada a **proceed** é implementada como uma chamada a uma implementação de hachura h_1, e, analogamente, em a_2 chama-se h_2. Como a_1 e a_2 chamam métodos de hachura diferentes, eles não são, a princípio, equivalentes. Entretanto, as hachuras h_1 e h_2 são equivalentes, pois foram geradas a partir de trechos de código idênticos da classe c. Logo, se h_1 e h_2 forem "unificadas" antes de a_1 e a_2, o algoritmo ADVICE-SHADOW-UNION, na segunda execução, identifica que as implementações a_1 e a_2 são réplicas.

Outro problema de ordem prática que surge na implementação de ADVICE-SHADOW-UNION é que o *bytecode* de classes é modificado mais de uma vez. Usando a biblioteca

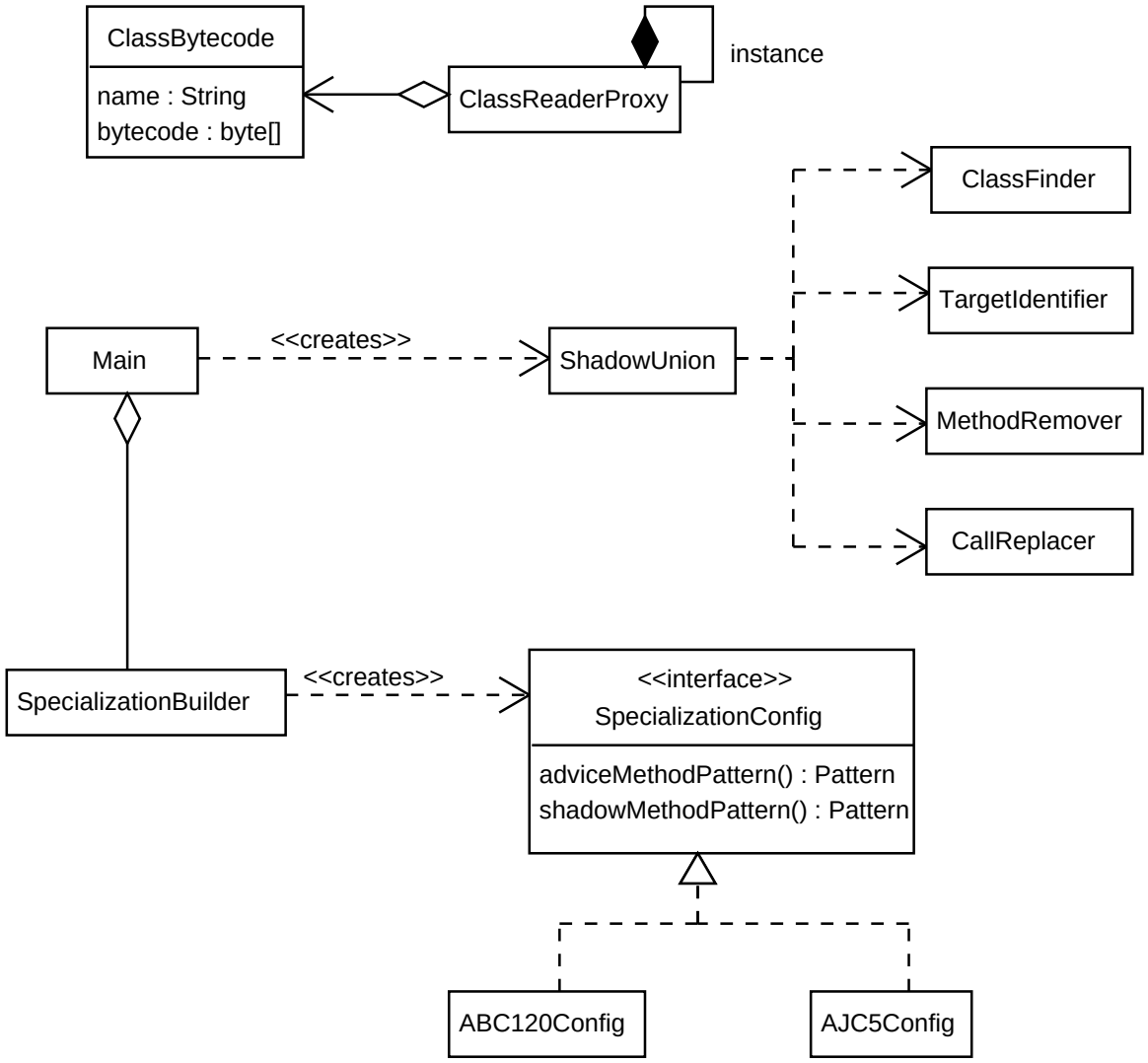

Figura 4.1: Classes principais da implementação de Advice-Shadow-Union

ASM, é possível ler um arquivo *class* e gerar um arranjo de *bytes* a partir do código nele contido. Para que o *bytecode* modificado por Remove-Methods seja posteriormente tratado por Replace-Calls, é necessário armazenar o *bytecode* resultante da execução do primeiro, para que esse código, e não o original, seja usado posteriormente. Para atender ao requisito de garantir que passos posteriores da execução do algoritmo modifiquem o *bytecode* já tratado em vez do original, criou-se a classe ClassReaderProxy.

ClassReader é uma classe da biblioteca ASM que pode ler *bytecode* a partir de um arquivo *class* ou de um arranjo de *bytes*. Todos os pontos do programa que necessitam de acesso ao *bytecode* de uma classe acessam um ClassReader a partir do *proxy*, que implementa o padrão de projeto *singleton* [GHJV95]. Na primeira vez que o

proxy é acessado para construir um ClassReader para uma classe c qualquer, o arquivo *"c.class"* que contém o *bytecode* daquela classe é carregado. Após qualquer modificação, o *proxy* é notificado e armazena os *bytes* resultantes, associados à classe c por meio de uma instância da classe ClassBytecode. Em chamadas seguintes para obtenção de um ClassReader para a classe c, o *proxy* o constrói, não a partir do arquivo *class* de c, mas sim do *bytecode* modificado que possui.

A relação de equivalência usada no algoritmo da Seção 4.1.2 consiste em uma família de classes de equivalência. O funcionamento de tal estrutura é ilustrado pelo diagrama da Figura 4.2: a família de classes de equivalência R contém m classes de equivalência, cujos elementos são equivalentes entre si de acordo com a relação E. Para que um novo elemento el_x seja introduzido nessa relação, realiza-se, para cada classe c_i com pivô $p[c_i]$, o teste $(p[c_i], el_x) \in E$, até que a classe c_x à qual el_x pertence seja encontrada. Para determinar se el_x é equivalente a todos os elementos de uma classe de equivalência, basta verificar se ele é equivalente ao seu pivô, já que toda relação de equivalência é transitiva. Note que, no momento de inclusão, el_x pode não pertencer a nenhuma das m classes de equivalência existentes, caso em que el_x é incluído em uma nova classe c_{m+1}. Na Figura 4.2, círculos são elementos do universo sobre o qual R é uma partição, e o losango E é a relação de equivalência que produz a partição R. Elementos destacados são pivôs de suas classes de equivalência.

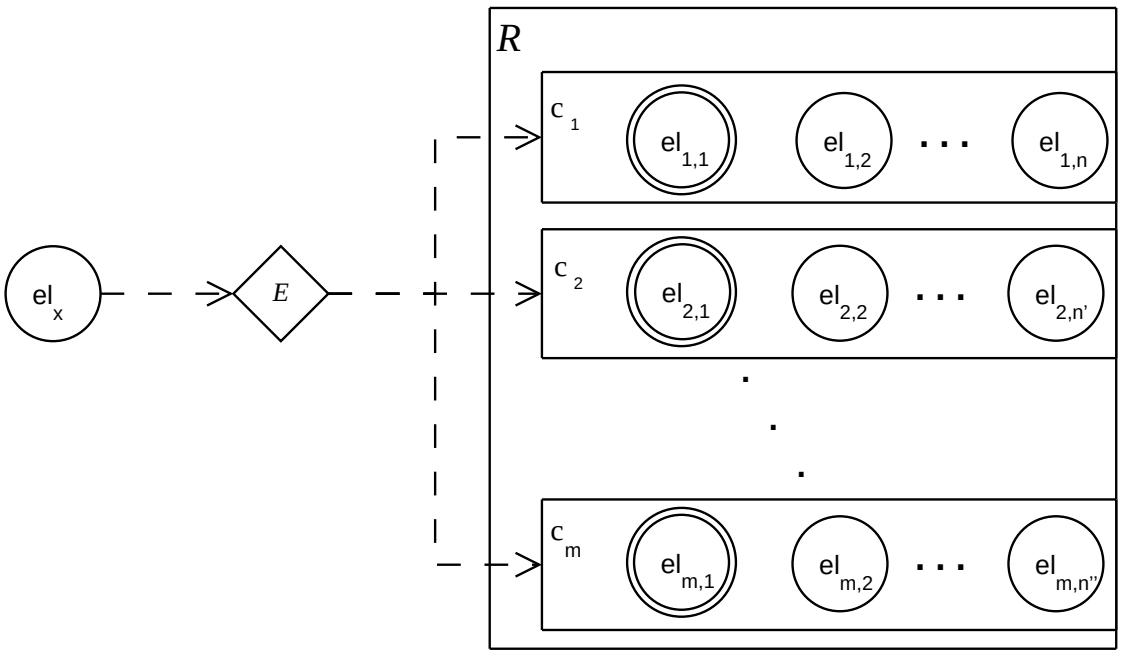

Figura 4.2: Estrutura que implementa uma relação de equivalência

Essa estrutura foi implementada, em Java, por meio das classes mostradas no diagrama da Figura 4.3. A interface Relation define a assinatura da operação de equi-

valência entre dois objetos do tipo parametrizado ObjectType. MethodEquivalenceRelation é uma implementação dessa interface para objetos do tipo MethodNode, que são representações de método da biblioteca ASM[4]. A implementação fornecida por essa classe do método isEquivalent realiza, usando a biblioteca ASM, as verificações de equivalência entre dois métodos, de acordo com a Definição 1 da Página 73.

A classe EquivalenceRelation define a estrutura da Figura 4.2, e implementa os procedimentos Pivot-For-Equiv-Class, Pivot-For-Element e Add-To-Equiv-Class, usados no algoritmo Advice-Shadow-Union como pontos de extensão. A inserção de elementos dessa implementação produz um *hash* a partir de métodos, e a partir desse *hash* encontra a classe de equivalência adequada para eles. Como a escolha de um pivô para uma classe é arbitrária, desde que consistente para todas as chamadas, o primeiro elemento de uma classe é retornado em chamadas a getPivot(List<ElementType >). A implementação de getPivot(ElementType) apenas encontra a classe de equivalência do elemento recebido e retorna o resultado de getPivot(List<ElementType>) para essa classe.

4.2 Integração aos Compiladores *ajc* e *abc*

4.2.1 Reaproveitamento de Métodos – *ajc*

No código gerado por compiladores de AspectJ, classes onde várias aplicações de adendos de contorno aparecem podem apresentar, como resultado da costura de código, trechos de código idênticos. Em alguns casos, é possível eliminar tais repetições, produzindo código *bytecode* com a mesma semântica do original, porém menor.

Para cada aplicação *sh* de um adendo de contorno a em uma classe X, o compilador *ajc* cria dois métodos em X: um, chamado *shadow method*, implementa o código de *sh*. Outro, chamado *local advice method*, implementa o corpo de a, onde chamadas a **proceed** são substituídas por chamadas a *sh*. Suponha, entretanto, que todas as hachuras sh_i, $0 \leq i \leq n$, da classe X sejam idênticas. Nesse caso, n métodos idênticos que implementam *sh* serão introduzidos em X, bem como n implementações similares de a. As implementações repetidas de a são ditas *similares* porque em cada a_i a chamada a **proceed** é substituída pela implementação de *sh* correspondente, sh_i.

A idéia central do reaproveitamento de métodos durante a costura de um adendo a na classe X é, uma vez que métodos sh_0 e a_0 tenham sido implementados em X, evitar

[4] Como descrito na Seção 4.1.1, ASM evita construir representações de *bytecode*. Para comparação de métodos instrução-a-instrução, entretanto, representações de métodos são necessárias, e foram construídas por meio do pacote opcional *asm-tree*, que constrói uma árvore a partir do *bytecode* de uma classe.

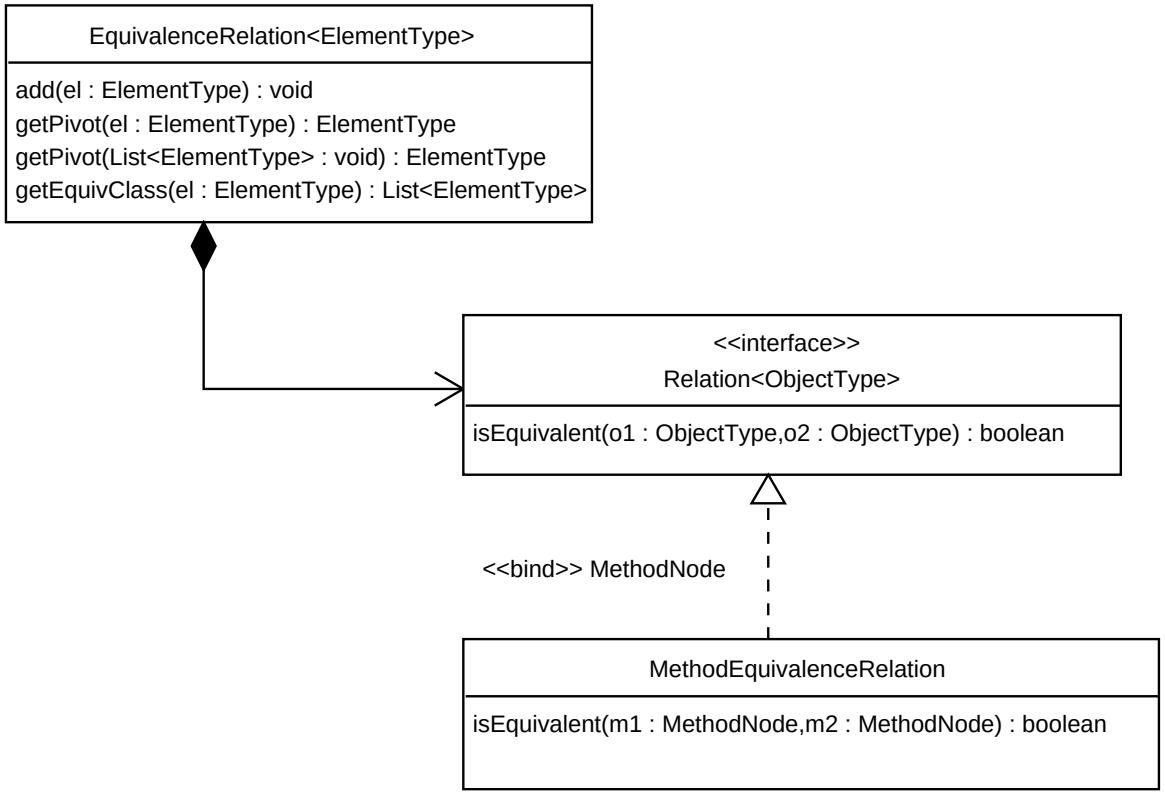

Figura 4.3: Implementação de estrutura de relação de equivalência

criar réplicas desses métodos, reutilizando os já existentes para implementar a costura desse adendo.

Todos os aspectos do programa são compilados como classes Java em uma etapa anterior à costura de código pelo *ajc*. Adendos são transformados em métodos dessa classe, e as instruções do corpo de cada adendo são implementadas, em *bytecode*, em seu respectivo método [HH04, Aspa].

No código do compilador *ajc*, como descrito na Seção 2.3, em aplicações de adendos de contorno que não contenham chamadas a **proceed** em classes anônimas, o método responsável por costurar código é BcelShadow.weaveAroundInline. Suponha novamente que a hachura em questão esteja em uma classe X. Esse método usa a representação em BCEL de X para adicionar um novo método a essa classe, sh, e copiar o código correspondente à hachura para o corpo desse método. Variáveis de contexto usadas na hachura são passadas por parâmetro para sh. Em seguida o combinador usa novamente a representação do *bytecode* de X para inserir um método a em X, copiando o código do corpo do adendo para o corpo de a. É importante observar que o método que implementa um adendo na classe que representa um aspecto usa *closures* para descrever o comportamento necessário à instrução **proceed**. Na cópia de instruções do adendo original para o método a, essa estratégia é substituída pela chamada ao método sh

recentemente criado.

O combinador do compilador *ajc* é implementado em um projeto com mais de 47000 linhas de código distribuídas em 369 classes. A documentação disponível para auxiliar na compreensão esse código, disponível em [Aspa], é superficial no sentido de que apresenta uma receita para se implementar uma nova construção na linguagem AspectJ, que identifica todas as modificações pontuais causadas por essa nova construção, mas não examina em profundidade as principais classes envolvidas no processo de costura ou o relacionamento entre elas.

Essa dificuldade em se modificar diretamente o código-fonte do combinador *ajc* levou a uma implementação em AspectJ do reaproveitamento de métodos. Laddad aponta, em [Lad03], que reaproveitamento de recursos[5] é um requisito transversal. É natural, portanto, que o reaproveitamento de métodos durante a costura de adendos de contorno seja implementado usando AspectJ.

Apenas um aspecto é suficiente para reaproveitar métodos criados durante a costura de adendos de contorno realizada pelo compilador *ajc*. O código simplificado da Listagem 4.1 mostra como o reaproveitamento é realizado em AspectJ para implementações de hachuras. As porções omitidas nessa listagem correspondem a implementações similares dos adendos listados para reaproveitamento de métodos.

O adendo posterior das linhas 35–42 preenche a tabela extractedMethods após chamadas a BcelShadow.extractMethod. O adendo de contorno das linhas 17–31 é usado para verificar se, para uma dada hachura *sh*, já existe um método que a implementa, e substituir, assim, a chamada a BcelShadow.extractMethod por uma busca na tabela extractedMethods. Estratégia similar é usada para implementações locais de adendos.

Representações em BCEL de métodos criados para hachuras e adendos são armazenadas em tabelas separadas. Métodos armazenados nessas tabelas são identificados unicamente pela concatenação do nome da classe em que a hachura ocorre com o nome do método que implementa o adendo aplicado, gerado pelo método makeKey.

```
1  public aspect ShadowUnionAspect {
2      pointcut weaveAroundInlineCalls(BcelAdvice munger) :
3          call(public void BcelShadow.weaveAroundInline(
4                  BcelAdvice, boolean))
5              && args(munger, ..);
6
7      pointcut extractMethodCalls(ShadowMunger munger,
8              BcelShadow enclosingShadow) :
```

[5] *pooling* e *caching*

```
9           call(public LazyMethodGen BcelShadow.extractMethod(
10                  String, int, ShadowMunger))
11              && args(.., munger)
12              && this(enclosingShadow);
13
14      Map/* String,LazyMethodGen */extractedMethods = new HashMap();
15      Map/* String,LazyMethodGen */adviceMethods = new HashMap();
16
17      LazyMethodGen around(ShadowMunger munger,
18                  BcelShadow enclosingShadow) :
19                  extractMethodCalls(munger, enclosingShadow)
20              && withincode(void BcelShadow.weaveAroundInline(
21                  BcelAdvice, boolean)) {
22          String key = makeKey(enclosingShadow, (BcelAdvice) munger);
23
24          if (extractedMethods.containsKey(key))
25              return (LazyMethodGen) extractedMethods.get(key);
26          else {
27              LazyMethodGen extractedMethod =
28                  proceed(munger, enclosingShadow);
29              return extractedMethod;
30          }
31      }
32
33      /* ... */
34
35      after(ShadowMunger munger, BcelShadow enclosingShadow)
36              returning(LazyMethodGen extractedMethod) :
37                  extractMethodCalls(munger, enclosingShadow)
38                  && withincode(void BcelShadow.weaveAroundInline(
39                          BcelAdvice, boolean)) {
40          String key = makeKey(enclosingShadow, (BcelAdvice) munger);
41          extractedMethods.put(key, extractedMethod);
42      }
43
44      /* ... */
45
46      private String makeKey(BcelShadow shadow, BcelAdvice munger) {
47          return shadow.getEnclosingClass().getClassName() + ":"
48                  + munger.getOriginalSignature();
49      }
50  }
```

Listagem 4.1: Implementação de um aspecto que reaproveita métodos criados para extração de hachuras durante a costura de código do compilador *ajc*

Para reaproveitar implementações de adendos, entretanto, uma modificação é necessária no código-fonte do método BcelShadow.weaveAroundInline: a criação de uma implementação local é composta de diversas tarefas, tais como criação de uma lista de parâmetros formais e especialização da instrução **proceed** para usar a implementação correspondente da hachura. No código original, essas ações são implementadas diretamente no corpo do método weaveAroundInline. Para simplificar a captura de todas essas ações por um único adendo, refatora-se esse código para condensar todas as ações relacionadas à criação de uma implementação local de adendo em um único método, chamado buildLocalAdviceMethod, que recebe todas as variáveis disponíveis em weaveAroundInline necessárias para construir a implementação local do adendo.

Essa aplicação circular de AspectJ na própria implementação da linguagem evidencia uma característica interessante da Programação Orientada por Aspectos: aspectos podem ser usados para estender, de forma modular, o comportamento de programas que não foram projetados com objetivo de extensibilidade. Uma vez que se possua domínio do código a ser modificado, aspectos podem ser usados para implementar novos requisitos ou modificar estratégias implementadas nesse código. O uso de AOP para estender o comportamento de programas existentes pode ser realizado também como uma fase prévia de estudo, na qual se identificam os pontos do código onde modificações são necessárias antes de realizar de fato a implementação modificando o código original.

A união de implementações repetidas de hachuras e adendos em classes onde ocorrem mais de uma hachura de um adendo de contorno não modifica a semântica de programas costurados. O aspecto de reaproveitamento de código foi compilado juntamente com o compilador *ajc*, e essa versão modificada do compilador passou nos 271 testes relacionados à costura de código distribuídos com o código fonte do compilador.

4.2.2 Etapa de Otimização – *abc*

O modelo de extensão do compilador *abc* determina que novas otimizações possam ser implementadas sem modificações ao seu código-fonte, por meio da introdução de transformadores de código escritos com o arcabouço Soot. Esse modelo é satisfatório quando as otimizações a serem introduzidas realizam modificações locais no código gerado, mas não modificações estruturais. As dificuldades causadas por essa restrição são descritas a seguir.

O arcabouço Soot de otimização de *bytecode* define um modelo de etapas, por meio das quais o código *bytecode* é modificado gradativamente. Para cada etapa de otimização existe uma representação intermediária correspondente. Por exemplo, cadeias de uso-definição são facilmente calculadas em código de três endereços e, por

isso, análises que produzem reaproveitamento de variáveis locais são implementados na etapa de otimização Jimple, que é uma representação desse tipo. Já a análise de *loads* e *stores* redundantes é realizada sobre a representação *Baf*, que é um código de pilha, onde essas operações são as mesmas do *bytecode* final gerado.

Entretanto, no compilador *abc* apenas dois tipos de otimização do Soot são ativadas: otimizações *peephole* e análises de fluxo. Nessas etapas, não é possível modificar a estrutura do programa implementado, e otimizações são restritas a manipular o corpo de métodos. Todavia, a união de implementações de adendos e hachuras depende, além da capacidade de se modificar o corpo de métodos, eliminar métodos de classes – isso não é possível nas etapas de otimização do arcabouço Soot proporcionadas pelo *abc* para extensões.

Uma alternativa para esse problema é introduzir o reaproveitamento de implementações de adendos e hachuras na costura de código do *abc*. Enquanto o autor estudava essa solução, entretanto, o problema foi relatado aos desenvolvedores desse compilador, que então informaram que esse problema já tinha sido solucionado para a próxima versão do *abc*. Optou-se então por interromper a integração do algoritmo proposto para união de adendos e hachuras e analisar o problema de repetição de variáveis de contexto, cuja solução é descrita no Capítulo 5.

A solução implementada pelos desenvolvedores do *abc* é o reaproveitamento de métodos, como descrito na Seção 4.2.1. Nessa estratégia, antes de criar implementações locais de adendos de contorno em uma classe A, o combinador do *abc* verifica se tal método já foi criado anteriormente para essa classe, reutilizando esse método em vez de criar uma réplica dele.

4.3 Resultados

Esta seção apresenta os resultados obtidos a partir da eliminação de implementação de adendos e hachuras, ou união de pontos de junção, de programas AspectJ. Apresentam-se uma avaliação empírica dessa otimização, baseada na sua aplicação a um pequeno *benchmark*, e um conjunto de equações que determinam, para um dado programa, o ganho que proporciona.

4.3.1 Avaliação Empírica

Alguns testes foram realizados para verificar, após o desenvolvimento da solução proposta, os ganhos obtidos com a eliminação de réplicas em programas com diferentes usos de adendos de contorno. Esta seção apresenta o *benchmark* utilizado e, em se-

guida, a redução no tamanho do código obtida para cada programa do *benchmark* pela execução do programa descrito na Seção 4.1.3.

Singleton

A implementação em AspectJ do padrão de projeto *singleton* [GHJV95], proposta por Hannemann e Kiczales em [HK02], utiliza adendos de contorno para substituir chamadas ao construtor de uma classe, e retornar a instância única criada na primeira ativação do construtor. O código-fonte de exemplo dessa implementação[6] realiza três chamadas em seqüência ao construtor de uma classe, para demonstrar que as referências retornadas por essas chamadas são iguais.

A implementação do padrão *singleton* em AspectJ, mostrada na Listagem 4.2, define um conjunto de junção abstrato, protectionExclusions, para permitir a exclusão da aplicação do aspecto em algumas classes, e uma interface vazia Singleton. O adendo de contorno definido nas linhas 5 a 12 captura chamadas a construtores de subclasses de Singleton, armazenando a primeira referência de cada classe tratada, e retornando esta referência nas demais chamadas ao construtor dessa classe.

```
1  public abstract aspect SingletonProtocol {
2      private Hashtable singletons = new Hashtable();
3      public interface Singleton {}
4      protected pointcut protectionExclusions();
5      Object around(): call((Singleton+).new(..))
6              && !protectionExclusions() {
7          Class singleton = thisJoinPoint.getSignature().getDeclaringType();
8          if (singletons.get(singleton) == null) {
9              singletons.put(singleton, proceed());
10         }
11         return singletons.get(singleton);
12     }
13 }
```

Listagem 4.2: Protocolo do padrão *singleton* em AspectJ, extraído de [HK02].

A classe *singleton* da aplicação de exemplo fornecida com o código-fonte dos padrões de projeto implementados por Hannemann é Printer, cujo código é mostrado na Listagem 4.3. PrinterSubclass, que também aparece nessa listagem, deve ser excluída do padrão *singleton*, ou seja, chamadas ao construtor de PrinterSubclass devem ser executadas sem interferência do aspecto.

[6] Disponível em http://www.cs.ubc.ca/~jan/AODPs/

```
1  public class Printer {
2      protected static int objectsSoFar = 0;
3      protected int id;
4      public Printer() {
5          id = ++ objectsSoFar;
6      }
7      public void print() {
8          System.out.println("\tMy ID is "+id);
9      }
10 }
11 public class PrinterSubclass extends Printer {
12     public PrinterSubclass() {
13         super();
14     }
15 }
```

Listagem 4.3: Classes de exemplo para aplicação do padrão *singleton*.

O aspecto concreto da Listagem 4.4 determina que Printer é um *singleton* e que PrinterSubclass deve ser ignorada pelo adendo de contorno de SingletonProtocol.

```
1  public aspect SingletonInstance extends SingletonProtocol {
2      declare parents: Printer implements Singleton;
3      protected pointcut protectionExclusions():
4          call((PrinterSubclass+).new(..));
5  }
```

Listagem 4.4: Implementação concreta do protocolo definido na Listagem 4.2.

O programa de teste desse padrão de projeto cria três instâncias de Printer e três instâncias de PrinterSubclass. Como mostram os testes da Listagem 4.5, as três referências de Printer são iguais, porém as referências para PrinterSubclass são diferentes. Note que, no método test1 da classe Main, realizam-se três chamadas ao construtor da classe Printer, que são capturadas pelo adendo de contorno de SingletonProtocol.

```
1  public class Main {
2      private static Printer printer1, printer2, printer3;
3      private static void test1() {
4          System.out.println("\nTest 1: All three printers should have the "
5           + "same ID");
```

```
6           printer1 = new Printer();
7           printer2 = new Printer();
8           printer3 = new Printer();
9           printer1.print();
10          printer2.print();
11          printer3.print();
12      }
13      private static void test2() {
14          System.out.println("\nTest 2: All three objects should "
15           + "be identical");
16          System.out.print("\tThey are ");
17          if ((printer1 == printer2) && (printer1 == printer3)) {
18              System.out.println("identical");
19          }
20          else {
21              System.out.println("not identical");
22          }
23      }
24      private static void test3() {
25          System.out.println("\nTest 3: Ensuring that subclasses can access "
26           + " the constructor");
27          System.out.println("(All three outputs should be different)");
28          printer1 = new PrinterSubclass();
29          printer2 = new PrinterSubclass();
30          printer3 = new PrinterSubclass();
31          printer1.print();
32          printer2.print();
33          printer3.print();
34      }
35      public static void main (String[] args) {
36          System.out.println("Testing SINGLETON pattern (aspectj) ...");
37          test1();
38          test2();
39          test3();
40          System.out.println("\n... done.");
41      }
42  }
```

Listagem 4.5: Programa principal.

SpaceWar

O jogo SpaceWar é um exemplo de AspectJ, distribuído com o pacote de ferramentas para desenvolvimento de programas em AspectJ com a IDE Eclipse, *AspectJ Development Tools*[7] (AJDT). Esse exemplo demonstra vários tipos de aplicações de AspectJ, e, dentre eles, a garantia de um contrato simples na interação com o usuário: uma nave destruída não pode ser movimentada.

Apenas um adendo de contorno é utilizado no código-fonte dessa aplicação, e seu código é mostrado na Listagem 4.6. O conjunto de junção referenciado na linha 2 do aspecto EnsureShipIsAlive, declarado na classe Ship, é[8]:

```
1 pointcut helmCommandsCut(Ship ship):
2     target(ship) && ( call(void rotate(int)) ||
3           call(void thrust(boolean)) ||
4           call(void fire()) );
```

Embora o código do adendo implementado em EnsureShipIsAlive seja pequeno, ele é aplicado em vários pontos das classes Player e Robot, que tratam comandos de jogadores humanos e virtuais, respectivamente.

```
1 aspect EnsureShipIsAlive {
2     void around (Ship ship): Ship.helmCommandsCut(ship) {
3         if ( ship.isAlive() ) {
4             proceed(ship);
5         }
6     }
7 }
```

Listagem 4.6: Garantia de contrato: uma nave deve estar "viva" para ser movimentada.

Rin'G e *Thread Safety*

Rin'G é um ambiente para execução e animação de algoritmos em grafos [CSST04], que usa a biblioteca Swing para interação com o usuário. Nessa biblioteca, todas as operações sobre componentes visuais devem ser realizadas na chamada *event-dispatching*

[7] Disponível em http://www.eclipse.org/ajdt

[8] Nesse exemplo um conjunto de junção é declarado em uma classe do programa (Ship) e usado, no corpo de um aspecto, para definir adendo.

thread, ou *thread* de despacho de eventos [Mic00]. Laddad, em [Lad03, Cap. 9], identifica a garantia dessa propriedade em um programa como um requisito transversal e oferece uma solução em AspectJ para esse problema.

O aspecto definido por Laddad garante que quaisquer chamadas a métodos da biblioteca Swing são realizadas na *thread* adequada. Como esse aspecto é de propósito geral, ou seja, aplicável em qualquer programa Java que utilize a biblioteca, aplicou-se essa solução ao Rin'G. Como esse programa faz amplo uso de Swing, os dois adendos de contorno do aspecto de *thread safety* são aplicados em vários pontos do programa: há 500 hachuras dos adendos de *thread safety* espalhadas nas 83 classes do Ring'G.

Aplicação da União de Adendos e Hachuras

Cada um dos programas do *benchmark* foi compilado com *abc* e *ajc*, e o código *bytecode* resultante de cada compilação foi passado para o programa de eliminação de réplicas de adendos e hachuras. Os tamanhos desses programas foram calculados pela soma dos tamanhos dos arquivos gerados durante a compilação. Tabela 4.2 apresenta as medições obtidas.

Aplicação	Código Original (*bytes*)	Código Transformado (*bytes*)	**Redução** (%)
Singleton			
abc	8115	7539	**7.1**
ajc	17403	16667	**4.2**
SpaceWar			
abc	150869	145391	**3.9**
ajc	222446	215995	**2.9**
Rin'G			
abc	947179	805162	**15**
ajc	1212273	1001661	**17.4**

Tabela 4.2: Tamanho do código gerado pelos compiladores *ajc* e *abc* e otimizado pelo algoritmo proposto

Analisando a natureza dos programas que figuram nessa medição e das estratégias de compilação adotadas pelos compiladores *ajc* e *abc* pode-se entender melhor as diferenças entre os resultados da execução da união de implementações de adendos e hachuras.

O compilador *ajc* é incremental, ou seja, permite que programas escritos em AspectJ sejam compilados em várias etapas independentes. Para que isso seja possível, o *ajc* deve supor que o código que está compilando é incompleto e que mesmo métodos e atributos não usados não podem ser removidos do código gerado, pois tais estruturas podem ser usadas em iterações subseqüentes da compilação. Por esse motivo, código

gerado para aspectos pelo compilador *ajc* inclui implementações de todos os adendos. Durante a costura de código, o corpo de um adendo é especializado em aplicações desse adendo pelo processo de expansão descrito na Seção 2.2.1.1, que gera implementações de adendos e hachuras nas classes em que aplicações ocorrem.

Uma avaliação analítica dos resultados obtidos pela aplicação do algoritmo de união de implementações de adendos e hachuras é apresentada na Seção 4.3.2, onde se definem as equações para se determinar, a partir de um programa de entrada e do conjunto de aspectos a serem aplicados a ele, a redução no tamanho do *bytecode* a ser obtida pela aplicação do algoritmo.

4.3.2 Equações de Redução de *Bytecode*

Os dados apresentados na Seção 4.3.1 mostram a redução do tamanho do *bytecode* de um conjunto de programas AspectJ otimizados pela união de implementações de adendos e hachuras. O objetivo desta seção é construir, a partir do estudo da estrutura de código *bytecode*, equações que determinem, para um dado programa em AspectJ, o *limite superior* do tamanho do código obtido pela aplicação dos algoritmos descritos nas seções anteriores deste capítulo.

As equações apresentadas aqui calculam o limite superior do tamanho do código pois, para alguns casos, detalhes internos dos algoritmos de expansão adotados pelos compiladores, bem como decisões de implementação de bibliotecas de manipulação de *bytecode* levam a reduções menores a partir da eliminação de trechos de código. Esses fatores estão fora do escopo da otimização proposta e são, portanto ignorados. Nota-se, entretanto, que para algumas instâncias, as equações encontram de fato o tamanho exato do código otimizado.

Seção 4.3.2.1 descreve brevemente as estruturas representadas no *bytecode* relevantes para a análise desse problema. Essas estruturas são usadas nas equações da Seção 4.3.2.2.

4.3.2.1 Estrutura do *Bytecode*

A descrição do *bytecode* aqui apresentada é um resumo[9] daquela encontrada em [LY99, Cap. 4]. Um arquivo *class*, que representa uma classe Java, é uma seqüência de *bytes*. Listagem 4.7 mostra, em sintaxe similar à da linguagem C, a estrutura de uma classe representada em *bytecode*. A sintaxe das estruturas mostradas nessa seção é a mesma usada em [LY99], de onde as listagens que mostram a estrutura do *bytecode* foram extraídas. Os tipos `u1`, `u2` e `u4` indicam que o *item* é representado por um valor sem

[9] Vários elementos do código *bytecode* são ignorados nesta seção em prol da objetividade.

sinal de 1, 2 e 4 *bytes*, respectivamente. *Itens* são os campos de estruturas em código *bytecode*, como magic e constant_pool_count na Listagem 4.7.

```
1  ClassFile {
2    u4 magic;
3    u2 minor_version;
4    u2 major_version;
5    u2 constant_pool_count;
6    cp_info constant_pool[constant_pool_count-1];
7    u2 access_flags;
8    u2 this_class;
9    u2 super_class;
10   u2 interfaces_count;
11   u2 interfaces[interfaces_count];
12   u2 fields_count;
13   field_info fields[fields_count];
14   u2 methods_count;
15   method_info methods[methods_count];
16   u2 attributes_count;
17   attribute_info attributes[attributes_count];
18 }
```

Listagem 4.7: Estrutura de uma classe em *bytecode*.

Todo código *bytecode* possui um repositório de constantes[10]. Constantes como nomes de classes e métodos são armazenadas nesse repositório e acessadas por meio de um índice único. O repositório de constantes é mostrado na Linha 6 da Listagem 4.7. Entradas nessa estrutura são do tipo cp_info, cuja estrutura é mostrada na Listagem 4.8.

```
1  cp_info {
2    u1 tag;
3    u1 info[];
4  }
```

Listagem 4.8: Estrutura de uma entrada no repositório de constantes

O item tag de um elemento do repositório de constantes indica qual é o seu tipo, para que interpretadores do código *bytecode* possam identificá-lo. Por exemplo, constantes *string* são marcadas pelo item tag com valor CONSTANT_Utf8. A estrutura dessa

[10] Do inglês *Constant pool*.

constante é mostrada na Listagem 4.9. Outros tipos de constantes são apresentados ao longo desta seção; a lista completa de tipos de constantes do repositório *bytecode* encontra-se em [LY99, Cap. 4]. Apenas o campo tag é comum aos vários tipos de constante *bytecode*. Na definição da constante CONSTANT_Utf8, que os itens length e bytes correspondem ao item info da Listagem 4.8. Cada tipo de constante especializa a estrutura cp_info definindo um formato para o conteúdo desse item.

```
1 CONSTANT_Utf8_info {
2   u1 tag;
3   u2 length;
4   u1 bytes[length];
5 }
```

Listagem 4.9: Estrutura que representa constantes *string* em *bytecode*.

Campos e métodos de uma classe são armazenados nos itens fields e methods da estrutura ClassFile. Um método é representado pela estrutura method_info, mostrada na Listagem 4.10. Os itens name_index e descriptor_index dessa estrutura apontam para entradas do repositório de constantes que armazenam *strings*. A estrutura field_info, que descreve um campo, é exatamente igual a method_info. O descritor referenciado por uma estrutura field_info, entretanto, casa com a produção da variável *FieldType* da gramática da Listagem 4.4; descritores de métodos casam com a variável *MethodDescriptor*.

Descritores, como os apontados pelo campo descriptor_index da Listagem 4.10, são *strings* que descrevem o tipo de campos e métodos de uma classe, cuja forma é definida pela gramática da Listagem 4.4. Tabela 4.3 mostra exemplos de campos e métodos em Java e seus descritores *bytecode*; uma descrição completa dos tipos denotados por *strings* produzidos por essa gramática pode ser encontrada em [LY99].

```
1 method_info {
2   u2 access_flags;
3   u2 name_index;
4   u2 descriptor_index;
5   u2 attributes_count;
6   attribute_info attributes[attributes_count];
7 }
```

Listagem 4.10: Estrutura que representa um método da classe definida em *bytecode*.

Descritores e outros *strings* são armazenados no repositório de constantes em estruturas cp_info do tipo CONSTANT_Utf8. Constantes não são repetidas no repositório de uma classe em *bytecode*. Considere a classe C da Listagem 4.11. Os itens descriptor_index da estrutura field_info que representam, no *bytecode*, os campos as e bs dessa classe apontarão para o mesmo índice do repositório de constantes. A constante nessa posição do repositório é do tipo CONSTANT_Utf8, e denota o *string* "[F", que descreve um arranjo de valores do tipo *float*.

```
1 class C {
2     public float[] as;
3     public float[] bs;
4 }
```

Listagem 4.11: Exemplo de classe Java

FieldType	→	BaseType	
	|	ObjectType	
	|	ArrayType	
BaseType	→	B	– byte
	|	C	– char
	|	D	– double
	|	F	– float
	|	I	– int
	|	J	– long
	|	S	– short
	|	Z	– boolean
ObjectType	→	L<classname>;	– referência
ArrayType	→	[ComponentType	– array
ComponentType	→	FieldType	
MethodDescriptor	→	(ParameterDescriptor*)ReturnDescriptor	
ParameterDescriptor	→	FieldType	
ReturnDescriptor	→	FieldType	
	|	V	

Figura 4.4: Gramática de descritores de tipo de *bytecode*

Declaração em Java	Descritor de Tipo Correspondente
int [] indices	[I
java.awt.Graphics g	Ljava/awt/Graphics;
Thread foo(**int** i, **boolean**[] bs)	(I[Z)Ljava/lang/Thread;

Tabela 4.3: Exemplos de descritores para campos e métodos Java

Bytecode permite estender o comportamento da Máquina Virtual Java (JVM) por meio de *atributos*, que podem ser anexados a classes, campos, métodos e até a outros atributos[11]. Um atributo é representado em *bytecode* pela estrutura attribute_info, mostrada na Listagem 4.12. A especificação da JVM, [LY99], determina que alguns atributos são obrigatórios em elementos do *bytecode*, e que os demais devem ser ignorados por implementações de JVM que os desconheçam.

```
1 attribute_info {
2   u2 attribute_name_index;
3   u4 attribute_length;
4   u1 info[attribute_length];
5 }
```

Listagem 4.12: Estrutura que representa atributos em *bytecode*.

O atributo Code, anexado a métodos, é um exemplo de atributo obrigatório para métodos que possuem corpo, ou seja, métodos que não são nativos e abstratos. Ele contém informações do código de um método; sua estrutura pode ser vista na Listagem 4.13. Note que os campos a partir de max_stack, na Linha 4 da Listagem 4.13, correspondem ao campo info da estrutura attribute_info.

As últimas estruturas relevantes para a corrente discussão são as usadas para identificar chamadas de métodos. As instruções de chamada de método em *bytecode*, nomeadamente invokestatic, invokevirtual e invokespecial, referenciam entradas no repositório de constantes que descrevem um método; essas entradas são do tipo CONSTANT_Methodref, cuja estrutura é mostrada na Listagem 4.14. O item name_and_type_index dessa estrutura referencia uma constante do tipo CONSTANT_NameAndType, também mostrada na Listagem 4.14. Note que essa última estrutura também é utilizada para referenciar campos de classes para instruções que os acessam, como getfield e putfield.

[11] Atributos de estruturas *bytecode* não devem ser confundidos com campos de classes Java. Campos de classes fazem parte do programa, enquanto atributos *bytecode* contêm informações para a máquina virtual.

```
1  Code_attribute {
2    u2 attribute_name_index;
3    u4 attribute_length;
4    u2 max_stack;
5    u2 max_locals;
6    u4 code_length;
7    u1 code[code_length];
8    u2 exception_table_length;
9    { u2 start_pc;
10     u2 end_pc;
11     u2 handler_pc;
12     u2 catch_type;
13   } exception_table[exception_table_length];
14   u2 attributes_count;
15   attribute_info attributes[attributes_count];
16 }
```

Listagem 4.13: Estrutura que apresenta o atributo Code, associado a métodos em *bytecode*.

```
1  CONSTANT_Methodref_info {
2    u1 tag;
3    u2 class_index;
4    u2 name_and_type_index;
5  }
6
7  CONSTANT_NameAndType_info {
8    u1 tag;
9    u2 name_index;
10   u2 descriptor_index;
11 }
```

Listagem 4.14: Estruturas que representam referências a métodos em *bytecode*.

As estruturas do *bytecode* descritas nesta seção são afetadas no processo de eliminação de réplicas de implementações de adendos e hachuras em programas AspectJ. Essas estruturas são usadas nas equações definidas nesta seção para determinar o impacto causado no tamanho do código *bytecode* de um programa a partir dessa otimização.

4.3.2.2 Equações

Nesta seção apresenta-se um conjunto de equações para determinar o ganho resultante da otimização proposta no Capítulo 4, que consiste em eliminar implementações repetidas de adendos e hachuras, geradas pelos compiladores *ajc* e *abc* durante a costura de código de adendos de contorno. O ganho dessa otimização é obtido a partir da eliminação de estruturas e constantes usadas para descrever métodos redundantes, reduzindo assim o tamanho do código *bytecode* das classes do programa.

As equações desta seção usam apenas os adendos de contorno de um programa, ignorando os demais, pois essas são as construções da linguagem AspectJ para as quais os compiladores *ajc* e *abc* geram métodos desnecessários, como descrito na Seção 3.1. A notação definida a seguir é independente do compilador usado para gerar o código *bytecode* de um programa AspectJ – as equações apresentadas independem das estratégias de costura de código dos compiladores *ajc* e *abc*.

Notação

Considere um programa AspectJ como uma tupla $P = (C, A)$, onde C é um conjunto de classes e A é um conjunto de adendos de contorno. Esses conjuntos são denotados por $C[P]$ e $A[P]$. Os adendos de contorno de um programa, elementos de $A[P]$, podem estar declarados em diferentes aspectos; tal fato é irrelevante para a discussão corrente, pois o código *bytecode* gerado para implementar adendos de contorno é independente do aspecto em que são declarados.

Considere também as seguintes definições relacionadas à natureza de um programa AspectJ:

- $\gamma(m)$ denota o número de chamadas ao método m no programa P;
- $\beta(a)$ denota o tamanho, em *bytes*, da implementação do adendo de contorno $a \in A[P]$;
- $\alpha(a, c)$ denota o número de aplicações do adendo de contorno $a \in A[P]$ na classe $c \in C[P]$;
- $\sigma(P)$ denota o tamanho total do programa P, calculado pela soma do tamanho em *bytes* do código *bytecode* de todas as suas classes.

Definem-se, ainda, variáveis que denotam o tamanho de estruturas do *bytecode*:

- $\eta_m(x)$ denota o tamanho do método x;
- $\eta_c(y)$ denota o tamanho da constante y;

- $\eta_a(z)$ denota o tamanho do atributo z.

Essas estruturas foram definidas na Seção 4.3.2.1, e as equações para calcular $\eta_m(x)$, $\eta_c(y)$ e $\eta_a(z)$ são descritas a seguir.

Equações

Sejam $P = (C, A)$ um programa escrito em AspectJ e uma classe $c \in C[P]$ com $\alpha(a, c)$ aplicações do adendo de contorno $a \in A[P]$. Como descrito no Capítulo 4, os compiladores *ajc* e *abc* criam, no *bytecode* do programa P, $\alpha(a, c)$ implementações do adendo a e das hachuras a que esse adendo se aplica na classe c. Como descrito no Capítulo 2, uma hachura é o trecho do código *bytecode* capturado pela expressão de conjunto de junção de um adendo de contorno, que precisa ser movido para um método para a implementação da semântica do comando **proceed** da linguagem AspectJ.

Suponha ainda que o adendo de contorno a capture chamadas a um determinado método f qualquer. Então todas as chamadas a f realizadas no corpo da classe c têm hachuras idênticas, e, portanto, o código gerado para implementar essas hachuras é idêntico, e esses métodos são réplicas uns dos outros. Essa mesma repetição de código ocorre na implementação do corpo do adendo a.

A técnica proposta no Capítulo 4 elimina do programa P, para toda classe c e todo adendo de contorno a, as $\alpha(a, c) - 1$ implementações desnecessárias de adendos e hachuras, produzindo um programa P' cujo código *bytecode* é menor do que o *bytecode* original P, ou seja, $\sigma(P') \leq \sigma(P)$. Equação 4.1 determina $\sigma(P')$ como a subtração do tamanho do *bytecode* eliminado de $\sigma(P)$.

$$\sigma(P') = \sigma(P) - \sum_{a \in A[P]} \left\{ \sum_{c \in C[P]} [(\alpha(a, c) - 1) \cdot \beta(a)] \right\} \tag{4.1}$$

No compilador *ajc*, as implementações de adendos são incluídas nas classes em que suas hachuras ocorrem. No *abc*, todas as implementações de adendo são incluídas no *bytecode* da classe que representa o aspecto em que foram declarados. A equação apresentada nesta seção é independente do compilador AspectJ que gerou um dado código *bytecode*, pois ambos os compiladores em estudo geram o mesmo número de métodos, para implementação de adendos e hachuras, porém em pontos diferentes do código.

O valor de $\alpha(a, c)$ é obtido a partir do código-fonte do programa P. O tamanho de uma implementação do adendo a, $\beta(a)$, é dado pela Equação 4.2.

$$\beta(a) = \eta_m(adv[a]) + \eta_m(sha[a]) \tag{4.2}$$

onde $adv[a]$ é o método que implementa o adendo a, e $sha[a]$ é o método que implementa uma hachura de a[12]. Pela definição da estrutura de um método em código *bytecode* apresentada na Seção 4.3.2.1, determina-se a Equação 4.3 para o tamanho de um método x em *bytes*. Nessa equação, *attributes[x]* representa o campo attributes da representação em *bytecode* do método x, *methodref[x]* representa uma estrutura de referência a esse método e *name_index[x]* representa a estrutura que armazena o nome e o tipo de x.

$$\begin{aligned} \eta_m(x) &= 4 \cdot 2 + \eta_c(name\_index[x]) + \sum_{a \in attributes[x]} [\eta_c(a)] \\ &\quad + \gamma(x) \cdot \eta_c(methodref[x]) \end{aligned} \tag{4.3}$$

Na Equação 4.3, o valor constante $4 \cdot 2$ corresponde aos itens access_flags , name_index, descriptor_index e attributes_count , que, como mostrado na Listagem 4.10 (pág. 96), ocupam 2 *bytes* cada. Os itens name_index e descriptor_index contém índices do repositório de constantes. Embora o tamanho da constante que representa o tamanho do nome do método x, $\eta_c(name\_index[x])$, faça parte dessa equação, seu descritor é ignorado, pois os descritores de todas as implementações de um mesmo adendo ou de uma mesma hachura são iguais[13]. Cada classe que contém uma chamada a um método x possui, em seu repositório, uma constante do tipo CONSTANT_Methodref, usada pela instrução de chamada a x para referenciá-lo. Assim, o valor $\gamma(x) \cdot \eta_c(methodref[x])$ denota, na Equação 4.3, o número de *bytes* usados no programa P para referenciar o método x.

O tamanho, no *bytecode*, de uma constante, como name_index, descriptor_index e methodref, que aparecem na Equação 4.3, pode ser calculado pela Equação 4.4, onde y é a constante. Essa equação segue a estrutura cp_info , mostrada na Figura 4.8 (pág. 95). Os valores constantes 1 e 2, somados nessa equação, representam o tamanho em *bytes* dos campos tag e length respectivamente.

$$\eta_c(y) = 1 + 2 + length[y] \tag{4.4}$$

Descrevem-se a seguir exemplos de aplicação da Equação 4.1 a programas AspectJ.

[12] Todas as implementações de adendo e hachura, $adv[a]_i$ e $sha[a]_i$, em uma classe $C[P]$, são idênticas.

[13] Note que, ao se introduzir essa consideração na equação para cálculo de $\eta_m(x)$, restringe-se o escopo de sua aplicação a métodos de implementação de adendos e hachuras.

Medição e Experimentos

Para aplicar a Equação 4.1 a um programa AspectJ P, é preciso compilá-lo e medir o tamanho em *bytes* de P, $\sigma(P)$ e, para cada adendo de contorno $a \in A[P]$, o tamanho de sua implementação, $\beta(a)$. As informações usadas para representar $\beta(a)$ no *bytecode* de P são obtidas por meio de uma ferramenta de visualização de *bytecode*, chamada *jclasslib*[14], que representa graficamente a seqüência de *bytes* de código *bytecode* em um formato similar ao da representação usada em [LY99].

Os exemplos a seguir foram compilados por *ajc*, versão 5.0, e *abc*, versão 1.2.0; o tamanho do *bytecode* resultante de cada compilação foi calculado como a soma do tamanho de todos os arquivos *class* produzidos. O resultado da Equação 4.1 é um limite superior para o tamanho do código *bytecode* obtido pela a otimização descrita no Capítulo 4. A diferença entre o tamanho esperado e o obtido pela execução da otimização proposta neste texto, para alguns casos, é analisada ao longo desta seção.

Descreve-se a seguir, separadamente, o resultado da Equação 4.1 aplicada ao código *bytecode* produzido pelos compiladores *ajc* e *abc*. Tabela 4.4 mostra o tamanho do *bytecode* do programa de exemplo do padrão de projeto *Singleton* [GHJV95], implementado em AspectJ por Hanneman e Kiczales [HK02]. Esse programa é o mesmo apresentado na Seção 4.3. A tupla a seguir representa o programa *Singleton* na notação aqui proposta, onde a é o adendo de contorno declarado em SingletonProtocol.

$$Singleton = (\quad \{\texttt{Main}, \texttt{Printer}, \texttt{PrinterSubclass}\}, \\ \{a\})$$

O *bytecode* gerado pelo compilador *abc* é chamado $Singleton_{abc}$, e o gerado por *ajc*, $Singleton_{ajc}$. Os conjuntos de classes e aspectos desses diferentes códigos *bytecode* são os mesmos, pois o código-fonte AspectJ que os gerou é o mesmo.

Programa	Tamanho (*bytes*)
$Singleton_{abc}$	8115
$Singleton_{ajc}$	17403

Tabela 4.4: Tamanho do *bytecode* gerado pelos compiladores *ajc* e *abc* para o código da implementação do padrão *Singleton* de Hanneman [HK02]

Singleton$_{\mathbf{abc}}$ Analisando novamente o código das Listagens 4.2 a 4.5, nas páginas 89 a 90, calcula-se o número de aplicações do único adendo de contorno[15] desse programa, declarado em SingletonProtocol:

[14] Disponível em http://www.ej-technologies.com/products/jclasslib/overview.html

[15] É necessário dar nomes abstratos a adendos neste texto pois, em AspectJ, eles são anônimos.

$$\begin{aligned}
\alpha(a, Main) &= 3 \\
\alpha(a, Printer) &= 0 \\
\alpha(a, PrinterSubclass) &= 0
\end{aligned}$$

O tamanho de uma implementação de a, $\beta(a)$, é dado pela soma do tamanho do método que implementa o próprio adendo a ao tamanho do método que implementa sua hachura, como mostra a Equação 4.2 (pág. 101). Os tamanhos desses métodos, denotados por $\eta_m(adv[a])$ e $\eta_m(sha[a])$, foram calculados a partir da Equação 4.3, e são mostrados nas equações a seguir. Os valores que aparecem nestas equações foram obtidos por meio da ferramenta *jclasslib*.

$$\begin{aligned}
\eta_m(adv[a]) &= 4 \cdot 2 + \eta_c(name_index[adv[a]]) + \sum_{t \in attributes[adv[a]]} [\eta_c(t)] \\
&\quad + \gamma(adv[a]) \cdot \eta_c(methodref[adv[a]]) \\
&= 4 \cdot 2 + 20 + (6 + 136) + 1 \cdot (5 + 5 + 20) \\
&= 200
\end{aligned}$$

$$\begin{aligned}
\eta_m(sha[a]) &= 4 \cdot 2 + \eta_c(name_index[sha[a]]) + \sum_{t \in attributes[sha[a]]} [\eta_c(t)] \\
&\quad + \gamma(sha[a]) \cdot \eta_c(methodref[sha[a]]) \\
&= 4 \cdot 2 + 12 + (6 + 40) + 1 \cdot (5 + 5 + 12) \\
&= 88
\end{aligned}$$

O valor de $\beta(a)$, calculado a partir da Equação 4.2 e dos valores de $\eta_m(adv[a])$ e $\eta_m(sha[a])$ acima, é dado pela Equação a seguir:

$$\begin{aligned}
\beta(a) &= \eta_m(adv[a]) + \eta_m(sha[a]) \\
&= 200 + 88 \\
&= 288
\end{aligned}$$

Portanto, o tamanho do programa $Singleton_{abc}$ otimizado, $Singleton'_{abc}$, é dado pela aplicação direta da Equação 4.1, mostrada a seguir. Note que o valor de $\sigma(Singleton'_{abc})$ é o mesmo encontrado pela execução de fato da otimização, mostrado na

Tabela 4.2.

$$\begin{aligned}\sigma(Singleton'_{abc}) &= \sigma(Singleton_{abc}) \\ &\quad - \sum_{a \in A[Singleton_{abc}]} \left\{ \sum_{c \in C[Singleton_{abc}]} [(\alpha(a,c) - 1) \cdot \beta(a)] \right\} \\ &= 8115 - (2 \cdot 288) \\ &= 7539\end{aligned}$$

Singleton$_{\mathbf{ajc}}$ O código *bytecode* gerado pelo compilador *ajc*, como descrito na Seção 4.3.1, é maior do que o gerado pelo *abc*. Além do número de métodos gerados ser maior, os nomes dos métodos gerados também são maiores, de forma que as implementações de adendos de contorno e hachuras ocupam mais *bytes*, no *bytecode* de programas gerados pelo *ajc*, do que naquele gerado pelo *abc*. Essa diferença pode ser observada nas equações a seguir.

$$\begin{aligned}\eta_m(adv[a]) &= 4 \cdot 2 + \eta_c(name\_index[adv[a]]) + \sum_{t \in attributes[adv[a]]} [\eta_c(t)] \\ &\quad + \gamma(adv[a]) \cdot \eta_c(methodref[adv[a]]) \\ &= 4 \cdot 2 + 27 + (6 + 131) + 1 \cdot (5 + 5 + 27) \\ &= 209\end{aligned}$$

$$\begin{aligned}\eta_m(sha[a]) &= 4 \cdot 2 + \eta_c(name\_index[sha[a]]) + \sum_{t \in attributes[sha[a]]} [\eta_c(t)] \\ &\quad + \gamma(sha[a]) \cdot \eta_c(methodref[sha[a]]) \\ &= 4 \cdot 2 + 20 + (6 + 32) + 1 \cdot (5 + 5 + 20) \\ &= 96\end{aligned}$$

Uma implementação do adendo $a \in A[Singleton_{ajc}]$ é, portanto, maior do que a calculada para o *bytecode* gerado pelo compilador *abc*, como mostrado a seguir.

$$\begin{aligned}\beta(a) &= \eta_m(adv[a]) + \eta_m(sha[a]) \\ &= 209 + 96 \\ &= 305\end{aligned}$$

O tamanho esperado do programa $Singleton_{ajc}$ otimizado, $Singleton'_{ajc}$, é dado pela Equação 4.5. Note, entretanto, que o valor calculado por essa equação é diferente daquele apresentado na Tabela 4.2.

$$\begin{aligned}
\sigma(Singleton'_{ajc}) &= \sigma(Singleton_{ajc}) \\
&\quad - \sum_{a \in A[Singleton_{ajc}]} \left\{ \sum_{c \in C[Singleton_{ajc}]} [(\alpha(a,c) - 1) \cdot \beta(a)] \right\} \\
&= 17403 - (2 \cdot 305) \\
&= 16793
\end{aligned}$$

Analisando o código *bytecode* gerado pelo compilador *ajc* para o programa *Singleton* e o otimizado pelo programa descrito na Seção 4.1.3, foi possível perceber que o erro de 126 *bytes* da Equação 4.5 está em atributos para depuração gerados pelo compilador *ajc* e na forma com que eles são tratados pela biblioteca ASM.

Na especificação do código *bytecode* [LY99], definem-se os atributos LineNumberTable e LocalVariableTable, que podem ser acoplados a métodos. Depuradores de código que interagem com a JVM acessam esses atributos para exibir informações relevantes sobre o código em execução ao programador. O compilador *ajc* produz esses atributos, para alguns métodos, sem conteúdo. Um exemplo de método cujo atributo LocalVariableTable é vazio, em $Singleton_{ajc}$, é SingletonInstance.aspectOf(), que retorna a instância do aspecto. A biblioteca ASM, ao analisar o *bytecode* desses métodos, trata atributos vazios como não-existentes e, portanto, ignora-os ao gerar código.

As equações apresentadas nesta seção são independentes do compilador usado na geração do *bytecode* de um programa AspectJ. Para alguns métodos, o compilador *ajc* gera atributos sem conteúdo, que são ignorados pela biblioteca ASM e não aparecem no *bytecode* otimizado pelo programa descrito na Seção 4.1.3. Essa característica não é considerada nas equações descritas nesta seção que, conseqüentemente, na análise de código *bytecode* gerado pelo compilador *ajc*, calculam o limite superior do tamanho do código otimizado produzido pelas técnicas propostas neste texto. Não obstante, o valor encontrado na Equação 4.5 é apenas $0,7\%$ maior do que a medida apresentada na Tabela 4.2.

4.4 Conclusões

A chamada união de pontos de junção elimina implementações repetidas de adendos de contorno e suas hachuras de código gerado para programas AspectJ. Neste capítulo

apresentou-se um algoritmo que elimina tais implementações em programas compilados por *ajc*, versão 5.0, e *abc*, versão 1.2.0. Mostrou-se também como integrar esse algoritmo à etapa de costura de código dos compiladores *ajc*, para que se evite criar métodos repetidos já durante a compilação.

Devido a problemas com a arquitetura e as bibliotecas usadas pelo compilador *abc*, não foi viável integrar o algoritmo proposto a esse compilador. Todavia, durante o desenvolvimento deste trabalho, a equipe responsável pelo compilador *abc* também percebeu o problema de implementações repetidas de adendos e hachuras, e a próxima versão desse compilador contará com essa otimização.

Ao se avaliar os resultados obtidos com a união de pontos de junção, percebeu-se que a redução no tamanho do código *bytecode* de programas otimizados pode chegar a cerca de 17% do tamanho total do programa, quando um adendo de contorno é aplicado a várias classes de um programa. Mostrou-se também, por meio de um estudo da estrutura de código *bytecode*, as construções afetadas pela união de implementações de adendos e hachuras, e apresentou-se um conjunto de equações capaz de determinar, a partir das características de um programa AspectJ, a redução em *bytes* a ser proporcionada por essa otimização.

É importante observar que as equações descritas neste texto foram aplicadas a um pequeno conjunto de programas, com o objetivo de se determinar as estruturas do código *bytecode* eliminadas por essa otimização. Em um estudo crítico do ganho proporcionado, é necessário ainda simular o comportamento dessas equações para programas com diferentes características, determinando assim casos em que o maior e o menor ganho são obtidos.

A integração dessa otimização no compilador *ajc* foi proposta ao grupo de desenvolvedores do compilador *ajc* [Cor06a], e a discussão que se seguiu identificou uma situação em que a união de adendos e hachuras não pode ser aplicada. Quando informações para depuração (*debug*) são geradas pelo *ajc*, o atributo LineNumberTable, que associa instruções do *bytecode* a linhas do programa fonte em AspectJ, é diferente para as várias implementações de hachuras em uma classe. Isso acontece porque, embora as hachuras sejam idênticas, elas aparecem em lugares diferentes do código-fonte. Como uma instrução do código *bytecode* só pode estar associada a uma linha do código-fonte, essa otimização não pode ser aplicada quando informações para depuração são geradas. Deve-se ressaltar, entretanto, que a aplicação da união de adendos e hachuras, mesmo quando aplicada junto à geração de informações de depuração, não modifica a semântica do programa em execução, mas apenas inviabiliza sua depuração.

Embora intuitivamente essa otimização não modifique a semântica de programas AspectJ, este texto não fornece uma demonstração formal dessa propriedade. Um trabalho futuro consiste em elaborar uma formalização de programas AspectJ e, usando

essa formalização, demonstrar que a eliminação de implementações equivalentes de adendos mantém a semântica do programa.

Capítulo 5

Eliminação de Contexto Repetido

Quando um adendo de contorno captura variáveis de contexto por meio das cláusulas **this**, **target** ou **args**, o compilador gera implementações expandidas desse adendo que possuem em sua assinatura parâmetros não usados. Esses parâmetros são gerados pelo combinador para fornecer ao corpo do adendo e ao método que implementa a hachura associada a ele os valores necessários à correta execução da hachura, embora sejam também fornecidos a partir da captura explícita de contexto do programador.

A repetição de variáveis de contexto em implementações expandidas de adendos aumenta desnecessariamente o *bytecode* gerado na costura de adendos de contorno, e torna sua execução mais lenta, já que argumentos não usados são carregados na pilha de execução para toda chamada a uma implementação de adendo que contenha variáveis de contexto repetidas. Esses argumentos também ocupam espaço nos *frames* de implementações de adendos, aumentando o uso de memória de programas AspectJ. Neste capítulo apresentam-se uma proposta de solução para esse problema, implementada no compilador *ajc* e os resultados obtidos por essa otimização para um pequeno conjunto de programas.

5.1 Captura de Contexto no Compilador *ajc*

Figura 5.1 apresenta o processo de compilação e costura de código para implementar o adendo de contorno das linhas 11 a 13 da Listagem 3.10 (pág. 65).

Nessa figura pode-se observar que o *front-end* do compilador gera *bytecode* a partir do código-fonte do método `test3`. O combinador então modifica esse *bytecode*, extraindo a hachura do adendo – a chamada a `m3` realizada pela instrução `invokevirtual #41` – para o método `m3_aroundBody4`, e introduzindo no ponto em que ela aparece em `test3` a chamada ao adendo, realizada pela instrução `invokestatic #88`.

No *bytecode* do método `test3` antes da costura realizada pelo combinador, note que

as variáveis disponíveis no ambiente da hachura presente nesse método são a instância da classe C, que ocupa a posição 1 do *frame* e é carregada pela instrução aload_1, e os argumentos da chamada a m3. Esses valores são incluídos nas listas de argumentos necessários para o método da hachura e, conseqüentemente, para a expansão do adendo. O objeto em execução (**this**) é sempre incluído no ambiente disponível de hachuras que aparecem em métodos não-estáticos, mesmo quando não usado na hachura; ele aparece no *bytecode* do adendo, m3_aroundBody5$advice, como um parâmetro do tipo M. A necessidade desse objeto surge da compilação incremental do compilador *ajc*: métodos de implementação de hachura são estáticos e, portanto, não possuem uma referência para esse objeto. Se um adendo que se aplique a uma hachura já extraída for adicionado ao programa em uma compilação posterior, o objeto **this** deve estar disponível no frame do adendo para captura.

Como apresentado no diagrama da Figura 2.3 (pág. 48) que mostra os objetos da arquitetura do compilador *ajc* envolvidos na costura de um adendo de contorno, o método weaveAroundInline da classe BcelShadow é o responsável por, após a identificação de hachuras do adendo, gerar código para sua aplicação nesse ponto do código. Nesse método, identificam-se as variáveis de contexto necessárias para a execução da hachura, e acrescentam-se esses valores à lista de parâmetros da expansão do adendo gerada para a aplicação e para o método que acomoda a hachura.

Seção 5.2 descreve as modificações feitas no código-fonte do compilador *ajc*, especialmente no método BcelShadow.weaveAroundInline, para eliminar variáveis de contexto repetidas.

5.2 Modificações Realizadas no Compilador *ajc*

O compilador *ajc* cria separadamente as listas de parâmetros dos métodos que implementam a hachura e a expansão do adendo, falhando em identificar a interseção existente entre essas listas. Como solução para o problema de repetição de variáveis de contexto, modificou-se o código de implementação de adendos de contorno para que identifique interseções entre o ambiente de uma hachura e o contexto capturado pelo conjunto de junção aplicado a ela. Além disso, foi necessário também modificar os trechos do código que geram instruções para carregar os argumentos para esses métodos, para que o código de chamadas aos adendos seja consistente em relação a sua assinatura.

No código-fonte do compilador *ajc*, adendos são representados por instâncias da classe BcelAdvice, que contém informações como o conjunto de junção associado ao adendo, o tipo de adendo (anterior, posterior ou de contorno), as variáveis de contexto

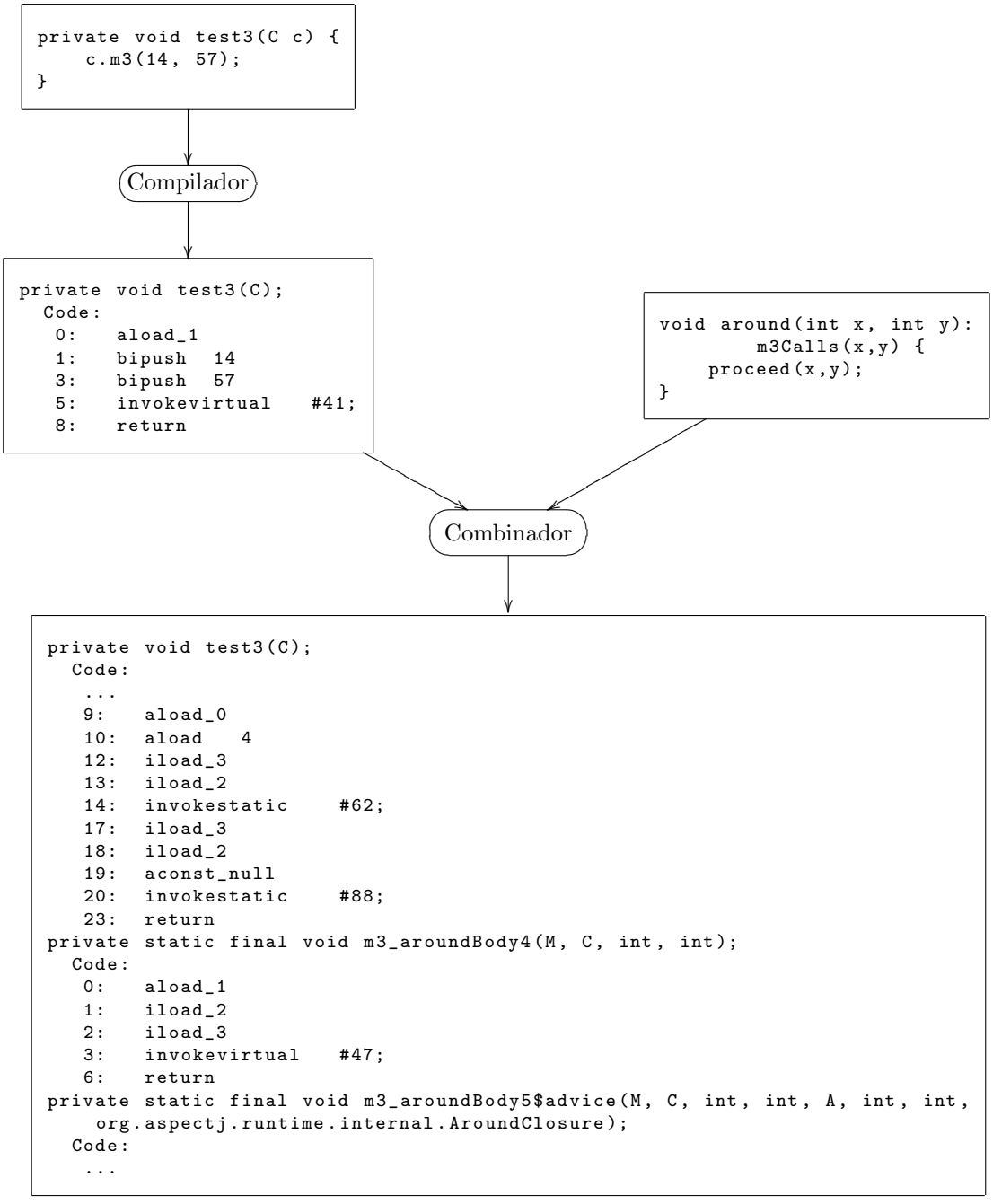

Figura 5.1: Geração de *bytecode* costurado a partir de código Java e AspectJ

expostas, além de seu código. Na solução proposta, durante a construção da lista de variáveis disponíveis no contexto da hachura, verifica-se na representação interna do adendo a presença de cada uma dessas variáveis. Toda variável de contexto disponível no contexto exposto pelo adendo é ignorada na criação da lista de parâmetros da expansão do adendo.

As listagens a seguir mostram trechos do método weaveAroundInline da classe Bcel–

Shadow relacionados à criação do método para implementação da hachura e da expansão do adendo de contorno para um novo método. O método da hachura é criado a partir da chamada a extractMethod, mostrada na Listagem 5.1, que move as instruções da hachura para um novo método criado na classe que a contém.

```
1 LazyMethodGen extractedMethod =
2             extractMethod(
3                 NameMangler.aroundCallbackMethodName(getSignature(),
                      getEnclosingClass()),
4                 Modifier.PRIVATE,
5                 munger);
```

Listagem 5.1: Criação do método que implementa a hachura.

Listagem 5.2 mostra a construção da lista de parâmetros do adendo expandido. Essas variáveis são identificadas como necessárias para implementar o corpo do adendo (argVarList) e para implementar a construção **proceed** (proceedVarList), que chama o método criado por extractMethod na Listagem 5.1. Os blocos da Listagem 5.2 tratam variáveis capturadas pelas cláusulas **this** (thisVar), **target** (targetVar) e **args** (argVars)[1]. As variáveis tratadas por esse trecho de código são elementos do *frame* do método que contém a hachura, e são representadas no código-fonte do compilador *ajc* por meio da classe BcelVar que armazena, dentre outras informações, o índice de uma variável no *frame* e seu tipo. Note, nessa listagem, que as mesmas variáveis são adicionadas às duas listas.

```
1  List argVarList      = new ArrayList();
2  List proceedVarList  = new ArrayList();
3  int extraParamOffset = 0;
4  if (thisVar != null) {
5      argVarList.add(thisVar);
6      proceedVarList.add(new BcelVar(thisVar.getType(), extraParamOffset));
7      extraParamOffset += thisVar.getType().getSize();
8  }
9  if (targetVar != null && targetVar != thisVar) {
10     argVarList.add(targetVar);
11     proceedVarList.add(new BcelVar(targetVar.getType(), extraParamOffset)
           );
12     extraParamOffset += targetVar.getType().getSize();
```

[1] Embora a variável **thisJoinPoint** apareça na Listagem 5.2, o uso dessa construção não causa captura de contexto repetido, pois ela não pode ser capturada em expressões de conjunto de junção.

```
13 }
14 for (int i = 0, len = getArgCount(); i < len; i++) {
15     argVarList.add(argVars[i]);
16     proceedVarList.add(new BcelVar(argVars[i].getType(), extraParamOffset
       ));
17     extraParamOffset += argVars[i].getType().getSize();
18 }
19 if (thisJoinPointVar != null) {
20     argVarList.add(thisJoinPointVar);
21     proceedVarList.add(new BcelVar(thisJoinPointVar.getType(),
       extraParamOffset));
22     extraParamOffset += thisJoinPointVar.getType().getSize();
23 }
```

Listagem 5.2: Criação da lista de parâmetros do adendo de contorno

A solução implementada para eliminar variáveis de contexto repetidas em código gerado pelo compilador *ajc* consiste em se verificar, antes da inclusão de uma variável na lista proceedVarList, se ela pertence ao contexto exposto pelo adendo. Listagem 5.3 mostra a obtenção do contexto exposto pelo adendo e a inclusão de variáveis capturadas pela cláusula **args** na assinatura do adendo, modificada para evitar repetição. Note que o trecho de código das linhas 6 a 12 corresponde ao código das linhas 14 a 18 da Listagem 5.2. O código modificado para variáveis capturadas pelas demais cláusulas de conjuntos de junção é similar.

```
1  java.util.HashSet<BcelVar> exposedState =
2      new java.util.HashSet<BcelVar>();
3  for (BcelVar var: munger.getExposedStateAsBcelVars(true))
4      exposedState.add(var);
5  /* ... */
6  for (int i = 0, len = getArgCount(); i < len; i++) {
7      argVarList.add(argVars[i]);
8      if (!exposedState.contains(argVars[i])) {
9          proceedVarList.add(new BcelVar(argVars[i].getType(),
               extraParamOffset));
10         extraParamOffset += argVars[i].getType().getSize();
11     }
12 }
13 /* ... */
```

Listagem 5.3: Uso do contexto exposto por um adendo para evitar captura repetida de variáveis

Outro trecho modificado para evitar captura de contexto é a criação da representação em BCEL do método de expansão do adendo. A lista de tipos que constituem a assinatura desse método é construída a partir da implementação original do adendo, gerada na classe que implementa o aspecto (nesse exemplo, a classe A – veja Listagem 3.10, pág. 65); copiam-se os parâmetros desse método e acrescentam-se à assinatura de sua expansão as variáveis necessárias para implementar a chamada à hachura. Na modificação realizada, acrescentam-se aos parâmetros da assinatura original apenas as variáveis que não pertencem ao contexto exposto do adendo, ou seja, apenas parâmetros que ainda não pertençam à sua assinatura.

A aplicação da otimização proposta neste capítulo a um conjunto de programas, e os resultados obtidos, são apresentados na Seção 5.4

5.3 Modificações Realizadas no Compilador *abc*

O problema de variáveis de contexto repetidas também aparece em código gerado pelo compilador *abc*. A associação de variáveis de contexto capturadas explicitamente pelo programador pelas cláusulas **args**, **this** e **target** é realizada pelo *front-end*. Para cada uma dessas associações, cria-se um resíduo dinâmico de ligação (*binding*), capaz de gerar uma instrução que carregue a variável associada como argumento para uma chamada ao adendo. Variáveis implicitamente necessárias para a execução de hachuras são associadas durante a extração da hachura para um método, realizada durante a costura.

Embora as associações dos dois tipos sejam realizadas em etapas diferentes, é possível, durante a extração da hachura, filtrar as variáveis já associadas no *front-end* a parâmetros do adendo. Para isso, basta identificar o contexto capturado pelo programador no adendo e filtrar o conjunto de variáveis do ambiente da hachura necessárias para a extração da hachura.

Como descrito na Seção 2.4, a costura de uma aplicação de adendo é realizada, no código-fonte do compilador *abc*, no método doWeave da classe AdviceApplicationInfo. Uma chamada realizada nesse método recupera o contexto necessário para a extração da hachura – variáveis locais usadas na hachura e definidas dela. O conjunto de variáveis de contexto calculado por essa chamada, context, é então usado para criar a lista de parâmetros das implementações do adendo e da hachura.

As ligações entre parâmetros do adendo, criadas pelas cláusulas de captura de contexto, e variáveis locais, podem ser recuperadas a partir do resíduo dinâmico da aplicação do adendo. Com essas ligações, pode-se finalmente filtrar o contexto necessário para a extração da hachura. O código da Listagem 5.4 consiste na filtragem

do conjunto de variáveis de contexto para eliminar repetições.

```
1  Residue.Bindings bindings = new Residue.Bindings();
2  adviceAppl.getResidue().getAdviceFormalBindings(bindings, null);
3  bindings.calculateBitMaskLayout();
4  for (Iterator contextIt = context.iterator(); contextIt.hasNext();) {
5      Local l = (Local) contextIt.next();
6      if (bindings.contains(l)) {
7          contextIt.remove();
8          shadowInternalLocalCount--;
9      }
10 }
```

Listagem 5.4: Modificação realizada no corpo do método doWeave da classe AdviceApplicationInfo.

O trecho de código dessa listagem deve ser introduzido no corpo do método doWeave da classe AdviceApplicationInfo após a extração da hachura para um método próprio. Essa necessidade ocorre porque a criação de instruções que carregam os valores de variáveis de contexto para a chamada **proceed** do adendo é realizada nesse ponto.

A implementação dessa otimização no código-fonte do *abc* mostra que o combinador desse compilador foi implementado de forma mais modular do que o *ajc*. A modificação necessária para filtrar o contexto capturado durante a extração da hachura é pontual, não causando efeitos colaterais em outros trechos do *abc*, ao contrário da modificação apresentada na Seção 5.2, em que vários trechos foram adaptados para atingir o mesmo resultado.

Embora essa implementação seja pequena e pontual, sua implementação mostra que o mecanismo de extensão do compilador *abc* é insuficiente para modificações no algoritmo de costura de código implementado. O desenvolvimento de tais modificações requer um estudo aprofundado da arquitetura e do código-fonte do combinador, que é composto de 61 classes e aproximadamente 6.000 linhas de código.

5.4 Resultados

A otimização proposta neste capítulo reduz o tamanho do código *bytecode* de classes, o tempo de execução e o uso de memória do programa que as contém. Nesta seção apresentam-se medidas da otimização obtida pela sua aplicação a um conjunto de programas AspectJ.

Como descrito previamente neste capítulo, na costura de adendos de contorno, a captura de contexto para implementação do comando **proceed** cria parâmetros repetidos

na assinatura de expansões desses adendos. A partir da captura repetida de variáveis de contexto surgem três problemas:

- o código *bytecode* de uma classe que contém hachuras do adendo de contorno contém estruturas redundantes;
- gasta-se tempo de execução carregando argumentos redundantes para execuções do adendo;
- o espaço de memória ocupado por ativações do adendo é maior do que necessário.

Assim como no Capítulo 4, apresentam-se nesta seção uma avaliação empírica dos resultados obtidos a partir da aplicação da otimização proposta neste capítulo a um pequeno *benchmark*, e, em seguida, um conjunto de equações capaz de determinar a redução no tamanho do código *bytecode* proporcionada por essa otimização para um dado programa AspectJ.

5.4.1 Avaliação Empírica

Apresenta-se a seguir o *benchmark* usado para avaliar o impacto da eliminação de variáveis de contexto repetidas em programas AspectJ. Todos os programas usam adendos de contorno que capturam contexto disponível em suas hachuras, de forma que tornam-se alvos da otimização proposta neste capítulo.

Embora a comparação entre o tamanho do código *bytecode* original e o otimizado seja direta, como apresentado a seguir na Seção 5.4.1.1, o impacto causado no tempo de execução de chamadas a um adendo de contorno é de difícil medição; as dificuldades encontradas na avaliação da redução no tempo de execução proporcionada pela eliminação de contexto repetido é apresentada na Seção 5.4.1.2.

Os programas apresentados a seguir, Linha de Produção e *SpaceWar*, são usados para mostrar os resultados obtidos a partir da eliminação de variáveis de contexto repetidas. Além disso, apresenta-se ainda um teste sintético em que se verifica o *speedup* obtido em chamadas a implementações de adendos otimizadas.

Linha de Produção: Memoização Transparente

Programação dinâmica é uma técnica usada na implementação de soluções para problemas que possuem *subestrutura ótima*, ou seja, problemas cujas soluções incluem soluções ótimas para seus subproblemas; Cormen e outros descrevem, em [CLRS02], uma metodologia para resolver problemas de programação dinâmica composta dos seguintes passos:

1. caracterizar a estrutura de uma solução ótima;
2. definir recursivamente o valor de uma solução ótima;
3. calcular o valor de uma solução ótima em um processo *bottom-up*;
4. construir uma solução ótima a partir de informações calculadas.

A eficiência de algoritmos de programação dinâmica está armazenar soluções para subproblemas quando são encontradas pela primeira vez, usando o valor já conhecido em vez de recálcula-lo em futuras ocorrências desses subproblemas; essa técnica é chamada *memoização*. A divisão do problema a ser resolvido, natural na solução recursiva criada no Passo 2 da metodologia apresentada acima, faz com que diversos caminhos da solução ótima levem aos mesmos subproblemas, o que a torna exponencial e, portanto, inviável para entradas grandes.

Entretanto, a solução recursiva é geralmente mais próxima da estrutura do problema, e mais legível, do que a solução desenvolvida a partir da construção *bottom-up* de uma solução ótima, como se pode observar pelos exemplos em [CLRS02, Cap. 15].

Usando AspectJ, é possível implementar memoização transparente na solução recursiva de problemas de programação dinâmica. Dessa forma, obtém-se um algoritmo eficiente e que, todavia, retrata a subestrutura ótima do problema. O problema da linha de produção é proposto em [CLRS02] como um problema de otimização em que se deseja encontrar o menor tempo possível para um automóvel ser construído em uma fábrica que possui duas linhas de produção paralelas, com diversas estações de latências diferentes.

Figura 5.2 mostra uma fábrica com duas linhas de produção. Cada linha de produção i é composta de n estações com latência $a_{i,k}$, $1 \leq k \leq n$. Um chassi pode ser transferido da linha de produção i para a j, gastando tempo $t_{i,k}$, onde k é a estação em que o chassi se encontra na linha i.

O algoritmo recursivo para encontrar o menor tempo possível para que um chassi passe por toda a linha de produção é apresentado em [CLRS02]. Listagem 5.5 apresenta sua implementação em Java. A classe ProductionLine representa uma linha de produção, e a correspondência entre seus atributos e os elementos da Figura 5.2 é mostrada em comentários nessa listagem. O arranjo stationLatencies corresponde à latência a da linha de produção; transferTimes corresponde ao tempo de transferência t; entryTime e exitTime corresponde aos tempos de entrada e e saída s, respectivamente; finalmente, n é o número de máquinas da linha de produção.

O método run da classe ProductionLineAlgorithm calcula a solução ótima para o problema como o mínimo entre o menor tempo para se passar pela última estação das linhas 1 e 2. O método recursivo fastest encontra o tempo mínimo para se passar

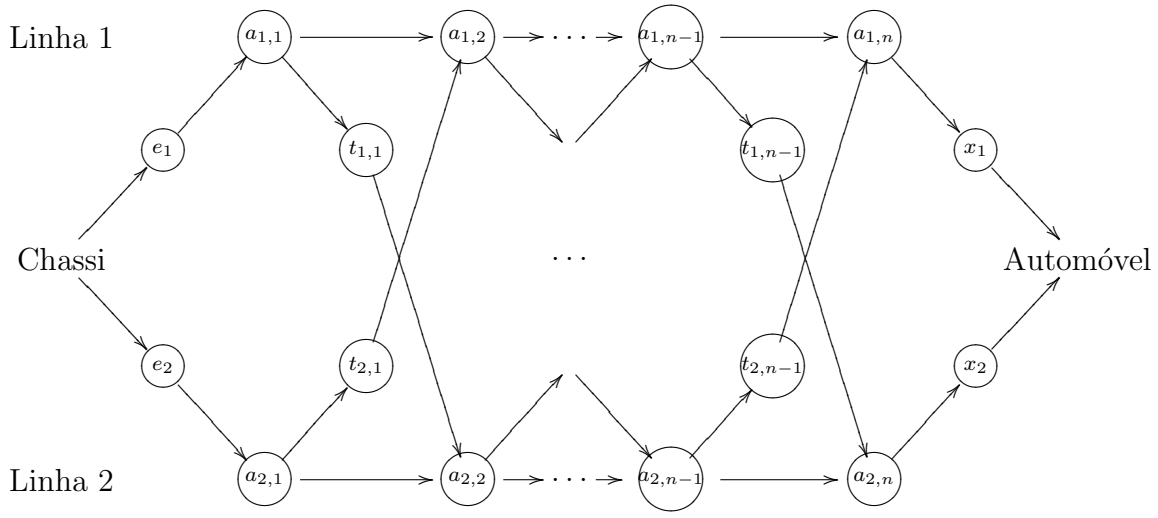

Figura 5.2: Linha de produção. Figura adaptada de [CLRS02].

por uma dada máquina j. Esse tempo é calculado como o mínimo entre passar pela máquina $j-1$ da a mesma linha de produção (line1) e passar pela uma máquina $j-1$ da outra linha (line2), gastando o tempo de transferência line2 . transferTimes [j −1].

O aspecto simples da Listagem 5.6, quando compilado junto à classe Production −LineAlgorithm, evita que soluções para subproblemas sejam recalculadas. Para isso, criou-se um adendo de contorno que captura chamadas ao método fastest e armazena o resultado de um subproblema quando encontrado pela primeira vez, retornando o valor já conhecido em chamadas subseqüentes.

```
1  public class ProductionLine {
2      protected int[] stationLatencies; // a
3      protected int[] transferTimes;    // t
4      protected int entryTime;          // e
5      protected int exitTime;           // x
6      protected int n;                  // n
7      /* ... */
8  }
9  public class ProductionLineAlgorithm {
10     public int run(ProductionLine line1, ProductionLine line2, int n) {
11         int t1 = fastest(n - 1,line1,line2) + line1.exitTime;
12         int t2 = fastest(n - 1,line2,line1) + line2.exitTime;
13         return Math.min(t1,t2);
14     }
15     private int fastest(int j, ProductionLine line1, ProductionLine line2) {
16         if (j == 0)
17             return line1.entryTime + line1.stationLatencies[0];
18         else {
19             int t1 = fastest(j - 1,line1,line2) + line1.stationLatencies[j];
20             int t2 = fastest(j - 1,line2,line1) + line2.transferTimes[j - 1]
21                                                 + line1.stationLatencies[j];
22             return Math.min(t1,t2);
23         }
24     }
25 }
```

Listagem 5.5: Implementação da solução recursiva para o problema da linha de produção.

```
1  public aspect MemoizationAspect {
2      ProductionLine line1;
3      ProductionLine line2;
4      int[] line1Cache;
5      int[] line2Cache;
6      pointcut algorithmInitialization(ProductionLine line1, ProductionLine line2,
7              int n):
8          call(int ProductionLineAlgorithm.run(..)) && args(line1,line2,n);
9      before(ProductionLine line1, ProductionLine line2, int n):
10         algorithmInitialization(line1,line2,n) {
11         line1Cache = new int[n];
12         line2Cache = new int[n];
13         this.line1 = line1;
14         this.line2 = line2;
15     }
16     pointcut fastestCalls(int j, ProductionLine line1, ProductionLine line2):
17         call(int ProductionLineAlgorithm.fastest(..))
18        && args(j,line1,line2);
19     int around(int j, ProductionLine line1, ProductionLine line2):
20         fastestCalls(j,line1,line2) {
21         int[] cache = getCache(line1);
22         if (cache[j] == 0) {
23             cache[j] = proceed(j,line1,line2);
24         }
25         return cache[j];
26     }
27     private int[] getCache(ProductionLine l) {
28         return l == line1 ? line1Cache : line2Cache;
29     }
30 }
```

Listagem 5.6: Aspecto de memoização.

SpaceWar

O jogo *SpaceWar*, já apresentado na Seção 4.3, é um exemplo da linguagem AspectJ, distribuído como parte do conjunto de ferramentas AJDT para desenvolvimento em AspectJ no ambiente Eclipse. Um adendo de contorno é usado, nesse programa, para garantir que movimentos de jogadores só podem ser executados se suas naves não tiverem sido destruídas. Esse aspecto é mostrado na Listagem 4.6 (pág. 92), e se aplica às classes que tratam movimentos de jogadores humanos, Player, e de oponentes virtuais, Robot. Ele captura o objeto-alvo de operações sobre a nave, instância de Ship.

O método da classe Player que trata operações do jogador é mostrado na Listagem 5.7; as hachuras do adendo da Listagem 4.6 nesse método são as chamadas a métodos da classe Ship das linhas 5, 8, 14 e 19.

```
1  public void keyPressed(KeyEvent e) {
2      int keyCode = e.getKeyCode();
3      boolean consumed = true;
4      if (keyCode == keyMapping.fire) {
5          ship.fire(); // atira
6      }
7      else if (keyCode == keyMapping.thrust && !thrust_on) {
8          ship.thrust(true); // acelera nave
9          thrust_on = true;
10     }
11     else if (keyCode == keyMapping.right &&
12                 rotation_direction != Ship.COUNTERCLOCKWISE) {
13         rotation_direction = Ship.CLOCKWISE;
14         ship.rotate(Ship.CLOCKWISE); // gira nave no sentido horário
15     }
16     else if (keyCode == keyMapping.left &&
17                 rotation_direction != Ship.CLOCKWISE) {
18         rotation_direction = Ship.COUNTERCLOCKWISE;
19         ship.rotate(Ship.COUNTERCLOCKWISE); gira no sentido anti-horário
20     }
21     else {
22         consumed = false;
23     }
24     if (consumed) e.consume();
25 }
```

Listagem 5.7: Método que trata movimentos do jogador no programa SpaceWar.

5.4.1.1 Redução no Tamanho do Código *Bytecode*

Uma análise baseada na estrutura do código *bytecode* dos resultados obtidos por essa otimização é apresentada na Seção 5.4.2. De forma simplificada, pode-se dizer que um

parâmetro p de um método m afeta o tamanho do código *bytecode* do programa das seguintes maneiras:

- a descrição da assinatura do método em *bytecode* inclui alguns *bytes* para referenciar o tipo de p;
- em chamadas a m, existem instruções para carregar os argumentos correspondentes a p.

Tabela 5.1 mostra o tamanho dos programas apresentados nesta seção, após compilados por duas versões dos compiladores *ajc* e *abc*: na segunda coluna, o código foi gerado pelas versões originais dos compiladores, e na segunda, por versões modificadas para eliminar variáveis de contexto repetidas durante a costura de adendos de contorno, como descrito na Seção 5.2.

Os programas Linha de Produção e SpaceWar apresentam diferentes percentuais de redução no tamanho do código *bytecode*. Essa diferença se deve à natureza dos programas: SpaceWar possui doze aplicações de um adendo de contorno divididas em duas classes, e Linha de Produção possui quatro aplicações em apenas uma classe. Entretanto, o código gerado para aplicações do adendo de contorno, em relação ao tamanho total do programa, é menor no programa SpaceWar do que em Linha de Produção e, por isso, a eliminação de variáveis de contexto repetidas tem impacto maior no *bytecode* do último.

A medida mostrada na Tabela 5.1 do código *bytecode* gerado pelo *abc* para o programa Linha de Produção parece destoar da medida para o *ajc*. Ao se comparar o código original e o código otimizado, percebe-se que, como efeito colateral da eliminação de variáveis de contexto repetidas, o *abc* consegue eliminar métodos repetidos de implementação de adendos e hachuras. Após a costura de código, o expansor (*inliner*) do *abc* tenta identificar réplicas entre os métodos gerados. Como essa etapa é realizada sobre código Jimple, há variáveis locais criadas para ligar valores de contexto a parâmetros do adendo, que tornam as várias expansões diferentes. Quando variáveis de contexto repetidas são eliminadas, essas variáveis locais não aparecem no código Jimple, e o expansor consegue identificar que esses métodos são iguais[2].

[2] A identificação de igualdade entre métodos realizada pelo *abc* é baseada em representações em *string* de código Jimple, e não da semântica dos métodos.

Aplicação	*Bytecode* Original (*bytes*)	*Bytecode* Otimizado (*bytes*)	**Redução** (%)
Linha de Produção			
ajc	14693	14568	**0.9**
abc	7481	6802	**9**
SpaceWar			
ajc	222446	222310	**0.06**
abc	150881	150746	**0.09**

Tabela 5.1: Tamanho do código gerado original e modificado pela otimização proposta.

5.4.1.2 Redução no Tempo de Execução de Programas

Eliminar variáveis de contexto repetidas da assinatura de expansões de adendos de contorno diminui o tempo gasto em chamadas a esses métodos. Abordagens diferentes, baseadas na natureza dos programas que constituem o *benchmark* usado, foram adotadas para medir o ganho de desempenho obtido em cada um.

O programa Linha de Produção recebe como entrada uma instância do problema, como definida em [CLRS02], e encontra uma solução ótima para essa instância. Para verificar o ganho de desempenho obtido pela otimização proposta, executou-se esse programa para entradas aleatórias de diferentes tamanhos.

As medidas de tempo de execução apresentadas nesta seção foram obtidas a partir da execução do programa para entradas aleatórias de mesmo tamanho 33 vezes, o que, segundo [Tri98], garante que as médias amostrais obtidas são próximas da média populacional. O programa original da Listagem 5.5, sem o aspecto de memoização, não foi incluído nessa análise[3].

Como ressaltado em [Han05], a medição do tempo de execução de programas Java é difícil por aspectos que fogem ao controle do programador, tais como coleta de lixo e realocação de *heap*. Além disso, a granularidade do método comumente usado para medir intervalos de tempo com precisão de milissegundos em Java, `java.lang.System.currentTimeMilis`, é dependente do sistema operacional. Para contornar alguns desses problemas, adotou-se a política de chamar alguns milhares de vezes todos os métodos do programa antes de iniciar a medição, garantindo que compilações *just-in-time* tenham sido feitas. Além disso, a máquina virtual é inicializada com um tamanho de *heap* maior do que o necessário para a execução dos programas, garantindo que não há realocações durante a execução.

[3] Deve-se notar que o algoritmo da Listagem 5.5, sem memoização, é exponencial. Para entradas com 32 estações, o algoritmo demora mais de 2 minutos para terminar; para uma entrada de tamanho 64, o programa foi interrompido após 6 horas de execução sem ter produzido resultado.

A biblioteca *hrtlib*[4] oferece precisão de nanossegundos para medidas em Java. Ela usa, no sistema operacional Windows XP, uma interrupção que conta o número de ciclos de CPU desde que o computador foi ligado e, por isso, tem alta freqüência de atualização. As medidas apresentadas nesta seção foram obtidas por meio dessa biblioteca.

Tabelas 5.2 e 5.3 mostram, para os compiladores *ajc* e *abc* respectivamente, medidas do tempo de execução do programa Linha de Produção para entradas de diferentes tamanhos. A eliminação de variáveis de contexto repetidas nessa aplicação proporciona, em alguns casos, redução de mais de 48% no tempo de execução. Essa redução é causada pela eliminação de instruções de carga de argumentos em chamadas ao adendo.

Entrada (n° de estações)	Original – A (ms)	Otimizado – B (ms)	$(B/A) - 1$ (%)
10	4.434	2.297	**-48.20**
50	4.444	2.391	**-46.20**
100	4.856	2.508	**-48.35**
150	5.011	2.644	**-47.24**
200	5.012	2.771	**-44.71**
250	5.378	2.904	**-46.00**
300	5.326	3.019	**-43.32**
350	5.472	3.151	**-42.41**
400	5.967	3.764	**-36.92**
450	6.176	3.873	**-37.29**
500	6.397	3.991	**-37.61**
550	6.762	4.093	**-39.47**
600	7.050	4.199	**-40.44**
650	6.819	4.292	**-37.06**
700	6.814	4.418	**-35.16**
750	7.934	5.383	**-32.15**
800	9.148	6.282	**-31.33**
850	9.122	6.310	**-30.83**
900	9.248	6.372	**-31.10**
950	9.167	6.408	**-30.10**
967	9.363	6.399	**-31.66**
1017	–	6.472	–

Tabela 5.2: Tempos de execução médios do programa Linha de Produção para entradas aleatórias, quando compilado pelo *ajc*.

As linhas destacadas no fim dessas tabelas mostram a economia de memória proporcionada por essa otimização. No *ajc*, o programa original é incapaz de executar para entradas maiores do que 967 estações, enquanto a versão otimizada executa para entradas de tamanho 1017, ou seja, entradas 5.17% maiores do que a versão com variáveis

[4] Disponível em `http://www.javaworld.com/javaqa/2003-01/01-qa-0110-timing_p.html`.

de contexto repetidas. No *abc*, o programa sem otimização executa para entradas com 1230 estações, enquanto o otimizado executa para entradas de tamanho 1341, que são 9.02% maiores.

Entrada (n° de estações)	Original – A (ms)	Otimizado – B (ms)	$(B/A) - 1$ (%)
10	4.498	2.357	**-47.60**
50	4.349	2.238	**-48.54**
100	4.562	2.388	**-47.65**
150	5.057	2.411	**-52.32**
200	4.853	2.531	**-47.85**
250	4.658	2.638	**-43.37**
300	5.057	2.672	**-47.16**
350	5.456	2.785	**-48.96**
400	5.328	3.285	**-38.34**
450	5.429	3.367	**-37.98**
500	5.724	3.411	**-40.41**
550	5.682	3.437	**-39.51**
600	5.891	3.521	**-40.23**
650	6.086	3.546	**-41.74**
700	6.045	3.622	**-40.08**
750	7.587	4.778	**-37.03**
800	7.633	4.791	**-37.23**
850	7.868	4.844	**-38.43**
900	7.676	4.901	**-36.15**
950	7.794	4.850	**-37.77**
967	7.529	4.844	**-35.66**
1017	7.752	5.101	**-34.20**
1230	7.083	4.860	**-31.39**
1341	–	5.089	–

Tabela 5.3: Tempos de execução médios do programa Linha de Produção para entradas aleatórias, quando compilado pelo *abc*.

SpaceWar é um programa interativo, de forma que torna-se necessário medir o tempo de execução dos métodos afetados pelo aspecto da Listagem 4.6 em vez do tempo total de execução do programa. Para isso, introduziram-se cronômetros da biblioteca *hrtlib* nas chamadas realizadas no corpo do método keyPressed da classe Player aos métodos capturados pelo adendo de contorno. Tabela 5.4 os resultados obtidos.

Teste Sintético

Embora as medidas da Tabela 5.4 mostrem uma redução no tempo de execução total dos métodos afetados por um adendo de contorno, esses tempos incluem também o corpo dos próprios métodos. Como a eliminação de variáveis de contexto afeta apenas

Método	Original – A (ms)	Otimizado – B (ms)	$(B/A) - 1$ (%)
ajc			
fire	2.691	2.595	**-3.57**
rotate	0.0117	0.0113	**-3.42**
thrust	0.0125	0.0114	**-8.80**
abc			
fire	2.358	2.291	**-2.84**
rotate	0.0107	0.0104	**-2.80**
thrust	0.0138	0.0120	**-13.04**

Tabela 5.4: Tempo de execução médio de chamadas aos métodos afetados pelo adendo de contorno da Listagem 4.6.

a chamada a adendos de contorno, criou-se um teste que realiza chamadas a métodos que não possuem implementação.

Neste teste, implementaram-se seis métodos sem corpo, com diferentes assinaturas; esses métodos são mostrados na Listagem 5.8. A cada método aplicou-se um adendo de contorno diferente, para garantir que o efeito de dois adendos aplicados a um mesmo método não afete a interpretação dos dados. Os adendos usam diferentes cláusulas de captura de contexto, e suas implementações simplesmente executam o comando **proceed** com as variáveis de contexto capturadas.

Entretanto, o tempo de execução de métodos vazios é da ordem de nanossegundos, e medidas obtidas a partir de uma única chamada a esses métodos, mesmo com a biblioteca *hrtlib*, são imprecisas. Assim, chamou-se cada método 100000 vezes e coletou-se o tempo total de execução dessas chamadas. As medidas apresentadas na Tabela 5.5 mostram que há um pequeno ganho no tempo de execução de chamadas a métodos após a eliminação de variáveis de contexto repetidas. Essa redução é observável tanto em código gerado pelo *ajc* quanto pelo *abc*.

```
1 private void f1(int i) {}
2 private void f2(int i, int j){}
3 private void f3() {}
4 private void f4() {}
5 private void f5(Object o) {}
6 private void f6(Object o1, Object o2) {}
```

Listagem 5.8: Métodos sem corpo para avaliação da duração de uma chamada.

Contexto capturado	Original – A (ms)	Otimizado – B (ms)	$(B/A) - 1$ (%)
args(i)			
ajc	0.6571	0.5397	-17.87
abc	0.4199	0.4137	-1.04
args(i,j)			
ajc	0.9702	0.7322	-24.53
abc	0.4137	0.3537	-12.09
target(m)			
ajc	0.5311	0.4727	-10.97
abc	0.2945	0.2922	-1.44
this(m)			
ajc	0.5386	0.4827	-1.08
abc	0.2953	0.2922	-1.50
args(o)			
ajc	0.6554	0.5314	-18.92
abc	0.4238	0.4137	-2.38
args(o1,o2)			
ajc	0.9708	0.7216	-25.67
abc	0.4137	0.3548	-14.24

Tabela 5.5: Tempos de execução médios de chamadas com contexto capturado a métodos vazios.

5.4.1.3 Conclusões

A eliminação de variáveis de contexto repetidas em programas AspectJ reduz o tamanho de seu *bytecode* por uma razão proporcional ao número de aplicações de adendos de contorno e do número e tipo de variáveis capturadas por esses adendos.

A economia de memória proporcionada pela otimização proposta é também relevante, viabilizando, em alguns casos, a execução de programas AspectJ para entradas maiores do que suas versões não-otimizadas. Essa economia é especialmente importante em algoritmos recursivos, em que vários registros de ativação de adendos de contorno podem ocupar a pilha de execução simultaneamente.

O ganho obtido no tempo de execução de chamadas a adendos otimizados foi mostrado em três exemplos: no programa Linha de Produção, mostrou-se que a eliminação de variáveis de contexto repetidas pode produzir ganhos próximos a 50% quando aplicada a adendos em métodos recursivos. No programa *SpaceWar*, mostraram-se reduções no tempo de execução de métodos e chamadas a ele. Finalmente, com um teste sintético, mostrou-se que eliminação de variáveis repetidas da assinatura de implementações de adendos diminui o tempo de execução de chamadas a elas, pois diminui o número de instruções executadas para carregar seus argumentos.

O teste sintético aplicado tornou mais claro o *speedup* proporcionado pela eliminação

de variáveis de contexto, embora se deva notar que ele é de pequena ordem: o *speedup* medido nesse teste é da ordem de nanossegundos.

5.4.2 Equações de Redução de *Bytecode*

Esta seção apresenta equações que determinam o limite superior do tamanho do código *bytecode* otimizado pela eliminação de variáveis de contexto repetidas. Além das estruturas apresentadas na Seção 4.3.2.1, as equações propostas nesta seção dependem de outras construções presentes em código *bytecode*, que são apresentadas a seguir. Acrescentam-se também à notação da Seção 4.3.2 algumas variáveis necessárias para a análise corrente.

Durante a interpretação de código *bytecode*, a máquina virtual mantém uma pilha de chamadas para cada *thread* em execução, contendo registros de ativação de métodos. Um registro de ativação é empilhado quando um método é chamado, e permanece vivo até que esse método retorne normalmente ou lance uma exceção. Um dos componentes desse registro, o *frame*, contém as variáveis locais do método. Variáveis no *frame* são endereçadas por índices de 1 *byte*, e são carregadas na pilha de execução (que não deve ser confundida com a pilha de chamadas) por meio de instruções *load* e *store*. Essas instruções são:

- iload, lload, fload, dload, aload: carregam valores do *frame* na pilha. Essas instruções possuem um operando, de um *byte*, que é o endereço do *frame* de onde a variável deve ser carregada. O tipo do valor a ser carregado é dado pela letra antes da palavra *load* nesses mnemônicos: *'a'* para referências, *'f'* para *float*, *'d'* para *double*, *'l'* para *long*, e *'i'* para *int*, *char* e *byte*;

- iload_<n>, lload_<n>, fload_<n>, dload_<n>, aload_<n>: essas instruções são especializações otimizadas das instruções acima para algumas posições comuns do *frame*, e não possuem operandos. O valor de n, para qualquer uma dessas instruções, está no intervalo $[0, 3]$;

- instruções *store* correspondentes às anteriores para armazenar valores no *frame*, como istore, astore_<n>, e assim por diante.

Para a corrente análise, faz-se necessário acrescentar ao conjunto de variáveis da Seção 4.3.2 as seguintes definições:

- $\lambda(v, m)$: denota o tamanho, em *bytes*, de uma instrução para carregar ou armazenar no *frame* do método m a variável v;

- $\theta(a)$: denota o conjunto das variáveis de contexto capturadas pelo adendo de contorno a;

- $\mu(a)$: denota o número de instruções *load* e *store* no corpo do adendo a com índice i, tal que $3 < i \leq 3 + |\theta(a)|$; ou seja, essas são as instruções de *load* e *store* que precisam de um operando[5];

- $\tau(t)$: denota o tamanho, em *bytes*, de uma representação do tipo t em um *string* descritor[6].

Considerando-se as especializações para os índices no intervalo $[0, 3]$ de instruções de *load* e *store* em código *bytecode*, instruções para carregar variáveis no corpo de métodos podem ocupar um ou dois *bytes*, dependendo de seu índice. Seja $pos[v, frame[m]]$ a posição da variável v no frame do método m onde é definida. O tamanho de uma instrução que carregue a variável v no corpo do método m é, portanto:

$$\lambda(v, m) = \begin{cases} 1 \ byte, & \text{se } 0 \leq pos[v, frame[m]] \leq 3 \\ 2 \ bytes, & \text{se } pos[v, frame[m]] > 3 \end{cases} \tag{5.1}$$

Como descrito nas seções anteriores deste capítulo, a eliminação de variáveis de contexto repetidas de um adendo afeta, no *bytecode*, os descritores do método que o implementa e as chamadas a ele. Equação 5.2 descreve, a partir do tamanho do programa original, $\sigma(P)$, e do efeito dessa otimização no código *bytecode* de P, o tamanho do programa otimizado, $\sigma(P')$.

$$\sigma(P') = \sigma(P) - \sum_{\substack{c \in C[P] \\ a \in A[P]}} \left[\alpha(a, c) \cdot \mu(a) + \sum_{v \in \theta(a)} (\tau(v) + \alpha(a, c) \cdot \lambda(v, a)) \right] \tag{5.2}$$

Nessa equação, subtraem-se do código *bytecode* de P os *strings* de descrições de variáveis de contexto repetidas, que ocupam $\tau(v)$ *bytes*, e as instruções que carregam esses argumentos, para cada aplicação (chamada) do adendo, que ocupam $\alpha(a, c) \cdot \lambda(v, a)$ *bytes*.

O termo $\alpha(a, c) \cdot \mu(a)$ dessa equação denota o número de instruções de *load* e *store* transformadas em instruções especializadas de 1 *byte*, ou seja, sem operando, causadas pela remoção de variáveis da assinatura do adendo. Com a eliminação de variáveis de contexto repetidas do adendo a, instruções que, no método original, usavam posições i do seu *frame*, $3 < i \leq 3 + |\theta(a)|$ passam a usar as posições $(3 + |\theta(a)|) - i$. Portanto,

[5] Essa variável denota apenas as instruções de *load* e *store* que serão transformadas em instruções especializadas do tipo load_<n> após a eliminação de variáveis da assinatura do método em que aparecem.

[6] Descritores são *strings* que indicam o tipo de métodos, variáveis e atributos em código *bytecode*, e são apresentadas mais detalhadamente na Seção 4.3.2.1

para esses índices, instruções de *load* e *store* sem operando são usadas no *bytecode* do programa otimizado.

Apresenta-se a seguir a aplicação da Equação 5.2 aos programas do *benchmark* apresentado na Seção 5.4.1. Nesse estudo descreve-se ainda como os valores usados nessa equação são medidos em código *bytecode*.

Medição e Experimentos

Para se aplicar a Equação 5.2 a um programa AspectJ P, é necessário, assim como no estudo da Seção 4.3.2, compilar P e, em seguida, medir o tamanho em *bytes* do código *bytecode* gerado. Esse tamanho, $\sigma(P)$, é calculado como a soma dos tamanhos das classes em $C[P]$. Para a medição das demais variáveis envolvidas na Equação 5.2, usa-se a ferramenta *jclasslib*, que permite navegar pela estrutura do *bytecode* de classes.

Nesta seção, avalia-se novamente o *bytecode* dos programas *SpaceWar* e *Linha de Produção*. Os tamanhos desses programas, sem a eliminação de variáveis de contexto repetidas, são mostrados na segunda coluna da Tabela 5.1. O objetivo desta seção é, portanto, aplicar a Equação 5.2 a representações desses programas e alcançar os valores da terceira coluna da Tabela 5.1.

Linha de Produção$_{ajc}$ Esse programa é constituído de três classes e um adendo de contorno, declarado no aspecto MemoizationAspect (Listagem 5.6, página 120). Seja a o nome desse adendo; pela observação do código-fonte do programa, encontra-se o número de aplicações do adendo a em suas classes:

$$\begin{aligned} \alpha(a, \mathit{ProductionLineAlgorithm}) &= 4 \\ \alpha(a, \mathit{ProductionLineTest}) &= 0 \\ \alpha(a, \mathit{ProductionLine}) &= 0 \end{aligned}$$

O conjunto de variáveis capturadas por a é composto dos três argumentos de chamadas ao método fastest da classe ProductionLineAlgorithm:

$$\theta(a) = \{j, line1, line2\}$$

Tabela 5.6 mostra os tipos desses argumentos, seus descritores, e o tamanho em *bytes* das instruções que os carregam em chamadas a a realizadas em um método m.

Analisando o *bytecode* do corpo do método que implementa o adendo a, determina-se o número de instruções de *load* e *store* com índice maior do que 3 e menor que ou

Argumento (v)	Tipo	Descritor	$\tau(v)$	$\lambda(v,m)$
j	**int**	`I`	1	2
line1	ProductionLine	`LProductionLine;`	16	2
line2	ProductionLine	`LProductionLine;`	16	2

Tabela 5.6: Variáveis capturadas pelo adendo de contorno em MemoizationAspect.

igual a $|\theta(a)|$:

$$\mu(a) = 5$$

Finalmente, pode-se aplicar a Equação 5.2 aos dados do programa Linha de Produção original, obtendo-se o seu tamanho quando compilado sem variáveis de contexto repetidas. No desenvolvimento a seguir, m denota um método em que aparece uma chamada ao adendo a. Observando novamente o código-fonte da classe ProductionLineAlgorithm, verifica-se que m denota, de fato, os métodos run e fastest , onde aparecem hachuras do adendo a. No desenvolvimento a seguir, denota-se o programa Linha de Produção, por razões de espaço, simplesmente por LP.

$$\begin{aligned}
\sigma(LP'_{ajc}) &= \sigma(LP_{ajc}) - \sum_{\substack{c\in C[LP_{ajc}]\\ a\in A[LP_{ajc}]}} \left[\alpha(a,c)\cdot\mu(a) + \sum_{v\in\theta(a)} (\tau(v) + \alpha(a,c)\cdot\lambda(v,a))\right] \\
&= 14645 - [4\cdot 5 + (\tau(j) + 4\cdot\lambda(j,m)) + (\tau(line1) + 4\cdot\lambda(line1,m)) \\
&\quad +(\tau(line2) + 4\cdot\lambda(line2,m))] \\
&= 14645 - [20 + (1 + 4\cdot 2) + (16 + 4\cdot 2) + (16 + 4\cdot 2)] \\
&= 14568
\end{aligned}$$

Linha de Produção$_{abc}$ As variáveis necessárias à Equação 5.2 independentes do compilador possuem o mesmo valor para o programa Linha de Produção quando compilado com os compiladores *ajc* e *abc*. São elas:

$$\begin{aligned}
\alpha(a, ProductionLineAlgorithm) &= 4 \\
\alpha(a, ProductionLineTest) &= 0 \\
\alpha(a, ProductionLine) &= 0
\end{aligned}$$

$$\theta(a) = \{j, line1, line2\}$$

Embora o conjunto de variáveis de contexto do adendo a, $\theta(a)$, seja o mesmo para

o código gerado pelos dois compiladores, o código gerado para manipulá-los *pode ser* diferente. Nesse exemplo, entretanto, as instruções que carregam variáveis de contexto capturadas pelo adendo possuem o mesmo tamanho em ambos os compiladores, e os dados apresentados na Tabela 5.6 valem também para o *bytecode* gerado pelo *abc*. O corpo da implementação de adendo gerada *abc* é diferente da gerada pelo *ajc*. No código gerado pelo *abc* para esse programa, o número de instruções *load* e *store* com operandos é:

$$\mu(a) = 4$$

Com esses dados, pode-se determinar o limite superior para o tamanho do programa otimizado:

$$\begin{aligned}
\sigma(LP'_{abc}) &\leq \sigma(LP_{abc}) - \sum_{\substack{c \in C[LP_{abc}] \\ a \in A[LP_{abc}]}} \left[\alpha(a,c) \cdot \mu(a) + \sum_{v \in \theta(a)} (\tau(v) + \alpha(a,c) \cdot \lambda(v,a)) \right] \\
&\leq 7481 - [4 \cdot 4 + (\tau(j) + 4 \cdot \lambda(j,m)) + (\tau(line1) + 4 \cdot \lambda(line1,m)) \\
&\quad + (\tau(line2) + 4 \cdot \lambda(line2,m))] \\
&\leq 7481 - [16 + (1 + 4 \cdot 2) + (16 + 4 \cdot 2) + (16 + 4 \cdot 2)] \\
&\leq 7408
\end{aligned}$$

Note que, enquanto no código determinado pelo *ajc* é possível determinar o tamanho exato do código gerado sem variáveis de contexto repetidas, em código gerado pelo *abc* pode-se apenas determinar o tamanho máximo desse código. Como descrito na Seção 5.4.1, a eliminação de variáveis de contexto repetidas pode causar, como efeito colateral, a eliminação de implementações repetidas de adendos e hachuras. Não é viável, entretanto, determinar analiticamente quando esse efeito colateral de fato ocorre, já que ele depende da representação interna Jimple criada na costura de adendos, e essa estrutura não é acessível para usuários do compilador *abc*.

Assim, se métodos forem unificados como efeito colateral da eliminação de contexto repetido de um programa P_{abc}, o tamanho de P'_{abc} poderá ser menor do que o encontrado pela Equação 5.2. O tamanho do código produzido pelo *abc* para o programa Linha de Produção, por exemplo, é de 6802 *bytes*, o que é bem menor do que o valor encontrado nesta seção.

SpaceWar$_{ajc}$ Por natureza, a análise por meio de equações do efeito da otimização proposta sobre o programa *SpaceWar* é mais complexa do que a análise para o programa Linha de Produção. Enquanto o adendo de contorno do programa Linha de Produção,

definido em MemoizationAspect, captura apenas chamadas ao método fastest , o adendo de contorno usado no programa *SpaceWar* intercepta chamadas a três métodos diferentes da classe Ship. Essas hachuras aparecem em duas classes diferentes, Robot e Player.

O contexto disponível em cada uma dessas hachuras é diferente das demais, pois os métodos thrust e rotate, interceptados por esse adendo, possuem parâmetros, e o método fire , também interceptado, não possui parâmetros. Dessa forma, embora o mesmo adendo de contorno a seja aplicado a todas as 12 hachuras do programa *SpaceWar*, elas serão representadas, nessa discussão, por implementações diferentes desse adendo, que serão chamadas a_{fire}, a_{rotate} e a_{thrust}. Encontram-se, então, o número de aplicações de cada adendo no programa[7]:

$$\begin{aligned} \alpha(a_{fire}, Robot) &= 1 & \alpha(a_{fire}, Player) &= 1 \\ \alpha(a_{rotate}, Robot) &= 3 & \alpha(a_{rotate}, Player) &= 3 \\ \alpha(a_{thrust}, Robot) &= 2 & \alpha(a_{thrust}, Player) &= 2 \end{aligned}$$

O único argumento capturado pelo adendo de contorno definido no aspecto Ensure −ShipIsAlive é o objeto-alvo de chamadas aos métodos acima. O contexto capturado por esse adendo é, assim,

$$\theta(a) = \{ship\}$$

Esse objeto é do tipo Ship, cujo descritor é `"Lspacewar/Ship;"`, o que implica que $\tau(ship) = 15$. Esse parâmetro do adendo a ocupa posições no seu *frame* posteriores a 3, e, portanto, $\lambda(ship, m) = 2$, onde m denota os métodos onde chamadas a a aparecem.

Os números de instruções de *load* e *store* com índice i, $3 < i \leq |\theta(a)|$, para as várias implementações de a, são:

$$\begin{aligned} \mu(a_{fire}) &= 1 \\ \mu(a_{rotate}) &= 2 \\ \mu(a_{thrust}) &= 2 \end{aligned}$$

Na aplicação da Equação 5.2 a seguir, note que calcula-se o efeito da eliminação de contexto repetido em uma classe e multiplica-se esse efeito por dois, já que a aplicação

[7] Note que o programa *SpaceWar* possui 22 classes e 6 aspectos. Nessa análise consideram-se apenas as classes e aspectos envolvidas na costura de adendos de contorno.

do adendo em ambas as classes é idêntica. O programa *SpaceWar* é representado, no desenvolvimento a seguir, por SW. Novamente, o valor calculado por essa equação é o valor mostrado na Tabela 5.1.

$$\begin{aligned}
\sigma(SW'_{ajc}) &= \sigma(SW_{ajc}) - \sum_{\substack{c\in C[SW_{ajc}]\\ a\in A[SW_{ajc}]}} \left[\alpha(a,c)\cdot\mu(a) + \sum_{v\in\theta(a)} (\tau(v) + \alpha(a,c)\cdot\lambda(v,a))\right] \\
&= 222446 - 2\cdot\{[\alpha(a_{fire}, Player)\cdot(\tau(ship) + \alpha(a_{fire}, Player)\cdot\lambda(ship, m))] \\
&\quad +[\alpha(a_{thrust}, Player)\cdot(\tau(ship) + \alpha(a_{thrust}, Player)\cdot\lambda(ship, m))] \\
&\quad +[\alpha(a_{rotate}, Player)\cdot(\tau(ship) + \alpha(a_{rotate}, Player)\cdot\lambda(ship, m))]\} \\
&= 222446 - 2\cdot\{[1 + (15 + 2)] + [6 + (15 + 6)] + [4 + (15 + 4)]\} \\
&= 222310
\end{aligned}$$

SpaceWar$_{abc}$ Assim como no programa Linha de Produção, alguns dos dados usados na Equação 5.2 são independentes do compilador, e por isso, são iguais no código *bytecode* gerado por ambos. Os seguintes dados são idênticos aos apresentados para o programa $SpaceWar_{ajc}$:

$$\begin{aligned}
\alpha(a_{fire}, Robot) &= 1 & \alpha(a_{fire}, Player) &= 1 \\
\alpha(a_{rotate}, Robot) &= 3 & \alpha(a_{rotate}, Player) &= 3 \\
\alpha(a_{thrust}, Robot) &= 2 & \alpha(a_{thrust}, Player) &= 2
\end{aligned}$$

$$\theta(a) = \{ship\}$$

Métodos gerados pelo *abc*, entretanto, são diferentes dos gerados pelo *ajc*. As assinaturas das implementações de adendo criadas pelo *abc* possuem menos parâmetros do que as geradas pelo *ajc*, o que gera *frames* menores e menos instruções de *load* e *store* com índices maiores do que 3. Portanto a redução no código proporcionada pela eliminação de variáveis de contexto repetidas no corpo dos métodos, transformando instruções de *load* e *store* com operando em instruções de 1 *byte*, é menor no código gerado pelo *abc* do que no *ajc*. No código gerado pelo *abc* para o programa SpaceWar, não há instruções afetadas por essa otimização em implementações do adendo, como mostram as variáveis:

$$\begin{aligned}
\mu(a_{fire}) &= 0 \\
\mu(a_{rotate}) &= 0
\end{aligned}$$

$$\mu(a_{thrust}) \;=\; 0$$

O desenvolvimento a seguir mostra que o valor calculado pela Equação 5.2 é um pouco maior do que o tamanho do programa apresentado na Tabela 5.1. Essa diferença é devida a um *bug* existente no compilador *abc* relacionado à associação de instruções a linhas do programa fonte para depuração. Quando o combinador detecta que duas implementações de adendo são idênticas, elas são transformadas em apenas uma. Essa otimização, entretanto, torna o atributo LineNumberTable do método restante inconsistente para uma das hachuras, já que uma instrução não pode ser associada simultaneamente a duas linhas do código-fonte.

$$\begin{aligned}
\sigma(SW'_{abc}) &= \sigma(SW_{abc}) - \sum_{\substack{c\in C[SW_{abc}]\\ a\in A[SW_{abc}]}} \left[\alpha(a,c)\cdot\mu(a) + \sum_{v\in\theta(a)} (\tau(v) + \alpha(a,c)\cdot\lambda(v,a))\right] \\
&= 150881 - 2\cdot\{[\alpha(a_{fire}, Player)\cdot(\tau(ship) + \alpha(a_{fire}, Player)\cdot\lambda(ship, m))] \\
&\quad +[\alpha(a_{thrust}, Player)\cdot(\tau(ship) + \alpha(a_{thrust}, Player)\cdot\lambda(ship, m))] \\
&\quad +[\alpha(a_{rotate}, Player)\cdot(\tau(ship) + \alpha(a_{rotate}, Player)\cdot\lambda(ship, m))]\} \\
&= 150881 - 2\cdot\{[0 + (15 + 2)] + [0 + (15 + 6)] + [0 + (15 + 4)]\} \\
&= 150767
\end{aligned}$$

Nesse exemplo, o atributo LineNumberTable do *bytecode* gerado originalmente pelo *abc* e o otimizado para as várias implementações de adendo de contorno existentes são diferentes entre si. Deve-se notar, entretanto, que quando se subtraem as diferenças entre o atributo LineNumberTable das implementações de adendo geradas pelo *abc* original e o modificado, o valor encontrado é de 21 *bytes*, o que é exatamente igual à diferença entre o valor medido para o *bytecode* do programa, apresentado na Tabela 5.1, e o valor calculado nesta seção. Como essa característica foge do escopo da otimização proposta, a equação apresentada nesta seção calcula um limite superior do tamanho do código otimizado.

5.5 Conclusões

Durante a costura de adendos de contorno, os compiladores *ajc* e *abc* capturam separadamente variáveis de contexto disponíveis no ambiente de hachuras e variáveis explicitamente capturadas pelo programador. Essa separação faz surgir variáveis de

contexto repetidas na assinatura de implementações de adendos de contorno no código *bytecode* de programas AspectJ.

Ao se eliminar variáveis de contexto repetidas, reduz-se o tamanho de implementações de adendos de contorno e, conseqüentemente, dos programas em que aparecem. Métodos afetados por essa otimização têm o espaço de memória de seus registros de ativação na pilha de chamada reduzidos, o que diminui o consumo de memória de programas AspectJ. O efeito dessa redução no consumo de memória é mais perceptível quando adendos de contorno são aplicados a métodos recursivos, situação em que um grande número de ativações desses adendos coexistem na pilha de chamadas.

Programas otimizados pela eliminação de variáveis de contexto repetidas deixam de carregar os argumentos eliminados em chamadas a adendos de contorno, reduzindo assim seu tempo de execução. Embora esse *speedup* seja pequeno, da ordem de nanossegundos para uma única chamada ao adendo, o ganho obtido em aplicações de adendos de contorno a métodos recursivo pode ser próximo a 50%, como mostram as medidas de tempo para o programa Linha de Produção.

Capítulo 6

Conclusões

Neste texto, apresentou-se um estudo de técnicas de costura de código aplicadas por dois compiladores da linguagem AspectJ: o AspectJ Compiler, *ajc*, e o AspectBench Compiler, *abc*. Nesse estudo, identificaram-se problemas de repetição de código causados pela costura de adendos de contorno.

Implementações repetidas de adendos e hachuras aparecem no código *bytecode* produzido pelos compiladores *ajc* e *abc* quando uma classe contém várias hachuras idênticas de um adendo de contorno. Desenvolveu-se uma ferramenta que, dado o código *bytecode* de um programa compilado pelos compiladores *ajc*, Versão 5.0, e *abc*, Versão 1.2.0, elimina as implementações desnecessárias de adendos e hachuras desse programa. Essa otimização, chamada união de implementações de adendos e hachuras, ou união de pontos de junção, diminui o tamanho do código *bytecode* de programas AspectJ. Quando um adendo é aplicado várias vezes em uma mesma classe, a redução pode ser considerável em relação ao tamanho do programa.

A redução no tamanho do código proporcionada por essa otimização é maior quando um adendo de contorno substitui vários pontos de junção no programa. O programa Rin'G, descrito na Seção 4.3, possui 83 classes e 500 hachuras de adendos de contorno. A eliminação de adendos e hachuras repetidas no código gerado para esse programa proporcionou uma redução de 15% no tamanho do *bytecode* gerado pelo *abc*, e 17.4% no código gerado pelo *ajc*. Propôs-se ainda uma maneira de se integrar essa otimização à costura de código do compilador *ajc*, e essa proposta está atualmente sendo discutida com seus desenvolvedores.

Durante a captura de variáveis de contexto para implementar adendos de contorno e suas hachuras, algumas dessas variáveis são capturadas mais de uma vez. Esse problema, chamado repetição de variáveis de contexto, surge quando valores são explicitamente capturados pelo programador, por meio das cláusulas **args**, **target** ou **this** na definição de adendos de contorno. Apresentaram-se soluções para esse problema

integradas aos compiladores *ajc* e *abc*, que atuam diretamente na costura de código. A eliminação de variáveis de contexto repetidas causa, para programas que usam adendos de contorno, redução no tamanho do código *bytecode*, no uso de memória e uma pequena redução no tempo de execução. A economia de memória é mais relevante em programas onde adendos de contorno são aplicados em métodos recursivos, onde várias ativações de um adendo coexistem na pilha.

Essa economia de memória permitiu que programas recursivos otimizados executassem para entradas 5% maiores em código gerado pelo *ajc*, e 9% maiores em código gerado pelo *abc*. Esses valores são proporcionais ao número de variáveis de contexto capturadas explicitamente pelo programador em conjuntos de junção associados a adendos de contorno.

Durante a implementação dessa otimização, foi possível avaliar comparativamente a extensibilidade dos compiladores. Embora a solução tenha a mesma essência no *ajc* e no *abc*, a filtragem do conjunto de variáveis de contexto no primeiro causa vários efeitos colaterais, que devem ser tratados em diferentes trechos do código. Como *abc* é mais modular, a costura de código está espalhada por várias classes e métodos. Assim, embora a modificação realizada na sua costura seja pontual, a identificação do ponto de modificação é mais complexa do que no *ajc*.

Ao estudar o código gerado pelos compiladores da linguagem AspectJ, foi possível também detectar quais usos, ou idiomas [Lad03], causam maior impacto no desempenho de programas dessa linguagem. Construções relacionadas à transversalidade estática não produzem resíduo dinâmico, de forma que não afetam o desempenho de aplicações, e são, de fato, correspondentes à inserção manual de estruturas nas classes de um programa.

A costura de adendos anteriores e posteriores introduz resíduo dinâmico, porém esse resíduo corresponde apenas a testes de aplicação de adendos. Adendos de contorno introduzem novas chamadas de método, que causam uma pequena perda de desempenho na execução de programas devida à preparação de argumentos e criação de registros de ativação. Quando *closures* são necessários, uma parte considerável do tempo de execução de programas gerados pelo *ajc* é gasta com inicialização desses objetos. Resíduos dinâmicos que de fato causam grande impacto no tempo de execução de programas AspectJ são os introduzidos pela construção **cflow**, para manipulação da pilha de ativações de pontos de junção. Entretanto, praticamente todos os resíduos dinâmicos relacionados a conjuntos de junção do tipo **cflow** podem ser eliminados em tempo de compilação, o que torna o código gerado pelo *abc* para programas que usem essas construções mais eficientes do que código gerado para os mesmos programas pelo *ajc*.

Código gerado pelo compilador *abc* mostra claramente que as construções da lingua-

gem AspectJ não são inerentemente caras, mas sim implementadas de forma cara no compilador *ajc*. Em verdade, as construções podem ser implementadas eficientemente, como no compilador *abc*; essa não é, porém, a maior prioridade dos desenvolvedores do compilador *ajc*, que têm aplicado maiores esforços em velocidade de compilação e costura, além da introdução de costura em tempo de carregamento para AspectJ.

Este trabalho mostra também que a arquitetura do compilador *abc* náo é tão facilmente estendida para modificações no seu algoritmo de costura de código quanto para inclusão de novas construções na linguagem AspectJ e otimizações que analisem o comportamento dinâmico de código gerado. Extensões no algoritmo de costura do *abc* podem requerer modificações não-modulares, introduzidas no código-fonte do compilador. A falta de documentação disponível pode tornar essa tarefa complexa e demorada.

A principal contribuição deste trabalho é a identificação de dois problemas causados pela costura de adendos de contorno por compiladores da linguagem AspectJ. Soluções para esses problemas foram propostas aos responsáveis por esses compiladores [Cor06a, Cor06b, Cor06c]. Os ganhos produzidos por essas otimizações foram analisados empiricamente, com *benchmarks*, e com um estudo do seu impacto na estrutura do código *bytecode*.

Outra contribuição deste texto é a descrição das principais técnicas de compilação de construções da linguagem AspectJ. Realizou-se um estudo detalhado da arquitetura dos compiladores *ajc* e *abc*, e as estratégias de costura de código adotadas por ambos foram apresentadas por meio de exemplos em AspectJ acompanhados do código *bytecode* gerado para eles.

Como trabalhos futuros, enumera-se:

- formalizar a semântica de programas AspectJ que usam adendos de contorno para demonstrar formalmente que as otimizações propostas não a modificam;
- simular o comportamento das equações de otimização propostas para determinar melhores e piores casos de aplicação;
- desenvolver equações que descrevam a redução no tempo de execução e no consumo de memória proporcionada pela eliminação de variáveis de contexto repetidas;
- desenvolver um otimizador de código gerado pelo *ajc* que elimine as estruturas do código mantidas para compilação incremental, mas desnecessárias para a execução do programa;
- analisar a expansão direta do corpo do adendo e da hachura para aplicações de adendos de contorno em métodos recursivos.

Embora a linguagem AspectJ já seja usada em ambientes de produção no desenvolvimento de sistemas, este estudo mostra que pequenas otimizações ainda podem melhorar o desempenho de programas escritos nessa linguagem, o que indica que as técnicas de compilação para programas orientados por aspectos ainda não estão sedimentadas nos ambientes existentes.

Apêndice A

Termos Relacionados a AspectJ

Tabela A.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[1], 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 A.1: Termos em português para Programação Orientada por Aspectos

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

Apêndice B

Bytecode

Bytecode é um formato de código de pilha, em que a estrutura de um programa e suas instruções são armazenadas como uma seqüência de *bytes*. Várias listagens ao longo deste texto apresentam código *bytecode* produzido para classes Java diretamente e costuradas por compiladores da linguagem AspectJ. As estruturas que definem classes e seus membros são descritas detalhadamente nos Capítulos 4 e 5, quando se analisa o efeito de uma das otimizações propostas sobre elas. Este apêndice descreve o formato de representações textuais do corpo de métodos em código *bytecode*, cuja compreensão é necessária para leitores que não estejam habituados com essa linguagem. A especificação completa do formato é apresentada em [LY99].

O corpo de um método em código *bytecode* é uma seqüência de *bytes*. Um *byte* específico nessa seqüência pode representar o código de uma instrução ou de seus operandos. Operandos podem ser valores ou referências ao repositório de constantes, onde entidades como nomes de métodos e atributos de classes, nomes de classes e descritores de tipos são armazenadas. Valores intermediários de computações e objetos são armazenados na pilha de execução.

Durante a execução, o corpo de um método é avaliado em um *frame*, que contém as variáveis locais do método. Parâmetros são armazenados como variáveis locais nessa estrutura. As instruções *load* e *store* são as únicas que manipulam variáveis locais; as demais instruções lêem seus operandos e escrevem os resultados em uma pilha de execução.

Listagem B.1 mostra o corpo de um método em código *bytecode*. Note que as instruções são rotuladas, nessa representação textual, com seu índice na seqüência de *bytes* que representa o corpo do método. Por exemplo, a primeira instrução, `new #26`, ocupa 3 *bytes* nessa seqüência: o primeiro contém o código da instrução `new` e os demais contêm o valor *26*, que é o índice do nome de uma classe a ser instanciada.

Famílias de instruções realizam operações similares no código *bytecode*. Por exem-

plo, toda instrução cujo nome começa com `iload` carrega um valor inteiro de uma variável local para a pilha, e o operando dessas instruções é a posição do *frame* em que a variável se encontra. Similarmente, instruções com mnemônico iniciado com `aload` carregam referências, e existe uma família para cada tipo suportado pela máquina virtual.

```
public static void main(java.lang.String[]);
  Code:
    0:   new #26;
    3:   dup
    4:   iconst_0
    5:   bipush  20
    7:   iconst_5
    8:   invokespecial   #29;
    11:  astore_1
    12:  aload_1
    13:  bipush  30
    15:  invokevirtual   #33;
    18:  goto    34
    21:  astore_1
    22:  getstatic   #14;
    25:  ldc #35;
    27:  invokevirtual   #41;
    30:  aload_1
    31:  invokevirtual   #46;
    34:  return
  Exception table:
    from   to  target type
      0    21    21   Class java/lang/Exception
```

Listagem B.1: Exemplo de código *bytecode* para um método.

Blocos *try-catch* aparecem no código *bytecode* como uma tabela de exceções. No código da Listagem B.1, por exemplo, a tabela de exceções determina que toda exceção do tipo java.lang.Exception que ocorrer entre as instruções 0 e 21 será tratada a partir da instrução 21. Tabela B.1 descreve as principais instruções do código *bytecode*.

Operação	Exemplos	Significado
Carga de variável local	`iload`, `aload`	Carrega variável local cujo índice é indicado pelo primeiro operando
Armazenamento de valor	`istore`, `astore`	Armazena valor no topo da pilha na variável indexada pelo primeiro operando
Lançamento de exceção	`athrow`	Lança objeto de exceção no topo da pilha
Carregar constante na pilha	`bipush`, `iconst`	Carrega o valor de uma constante, que é o primeiro operando da instrução, na pilha
Operações aritméticas	`dadd`, `ldiv`, `fsub`, `icmpl`	Realiza operação aritmética sobre dois valores no topo da pilha, e empilha o resultado
Acessar variáveis de instância e de classe	`getfield`, `getstatic`	Coloca no topo da pilha valor do campo identificado pelo primeiro operando
Chamada de método	`invokeinterface`, `invokespecial`	Chama método definido pelo primeiro operando
Atribuir valor a variáveis de instância e de classe	`putfield`, `putstatic`	Atribui valor do topo da pilha ao campo identificado pelo primeiro operando
Retorno	`return`, `iterturn`	Retorna de um método, possivelmente empilhando um valor na pilha do chamador

Tabela B.1: Principais instruções do código *bytecode*.

Referências Bibliográficas

[ABCdg] Aspect Bench Compiler development group, *Official* abc *project page*.

[ACH+04] Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondrej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble, *Building the abc AspectJ compiler with Polyglot and Soot*, Tech. Report abc-2004-2, The abc Group, Outubro 2004.

[ACH+05a] ———, *abc: An Extensible AspectJ Compiler*, TAOSD'05 (2005).

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

[Aspa] AspectJ Team, *Guide for Developers of the AspectJ Compiler and Weaver*, Disponível no módulo *docs* do código-fonte do compilador *ajc*.

[Aspb] ———, *Página oficial do projeto AspectJ e do compilador* ajc, http://www.eclipse.org/aspectj, Última visita em Março de 2006.

[Asp05a] ———, *The AspectJ development environment guide*, 2005, http://www.eclipse.org/aspectj/doc/released/devguide/index.html.

[Asp05b] ———, *The AspectJ development kit developer's notebook*, 2005, http://www.eclipse.org/aspectj/doc/next/adk15notebook/index.html.

[Böl99] Kai Böllert, *On weaving aspects*, Lecture Notes in Computer Science, Springer-Verlag GmbH, 1999.

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

[CLRS02] Thomas Cormen, Charles Leiserson, Ronald Rivest, and Clifford Stein, *Algoritmos: Teoria e Prática*, Editora Campus, 2002, tradução da 2ª edição americana.

[Cor06a] Eduardo Santos Cordeiro, *Around advice weaving generates repeated methods*, *Bug report* disponível em `https://bugs.eclipse.org/bugs/show_bug.cgi?id=154253`, Agosto 2006.

[Cor06b] ———, *Around weaving produces repeated context variables*, *Bug report* disponível em `https://bugs.eclipse.org/bugs/show_bug.cgi?id=166064`, Novembro 2006.

[Cor06c] ———, *Around weaving produces repeated context variables*, *Bug report* disponível em `http://abc.comlab.ox.ac.uk/cgi-bin/bugzilla/show_bug.cgi?id=77`, Novembro 2006.

[CSST04] Eduardo Cordeiro, Italo Stefani, Tays Soares, and Fabio Tirelo, *Rin'g: Um ambiente não-intrusivo para animação de algoritmos em grafos*, XII WEI, em Anais do SBC 2004 - XXIV Congresso da Sociedade Brasileira de Computação, vol. 1, Julho 2004.

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

[DvZH03] Markus Dahm, Jason van Zyl, and Enver Haase, *Página oficial do projeto BCEL*, http://jakarta.apache.org/bcel, 2003, Última visita em Junho de 2006.

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

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha, *The Java Language Specification*, terceira ed., Addison-Wesley Professional, 2005, Disponível em http://java.sun.com/docs/books/jls/index.html.

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

[Han05] Stuart Hansen, *Interpreting java program runtimes*, SIGCSE '05: Proceedings of the 36th SIGCSE technical symposium on Computer science education (New York, NY, USA), ACM Press, 2005, pp. 36–40.

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

[HK02] Jan Hannemann and Gregor Kiczales, *Design pattern implementation in java and aspectj*, OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (New York, NY, USA), ACM Press, 2002, pp. 161–173.

[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).

[Kul05] Eugene Kuleshov, *Introduction to the ASM 2.0 bytecode framework*, Agosto 2005, http://asm.objectweb.org/doc/tutorial-asm-2.0.html.

[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*, segunda ed., Addison-Wesley Professional, 1999, Disponível em http://java.sun.com/docs/books/vmspec/index.html.

[Mic00] Sun Microsystems, *Threads and swing*, Setembro 2000, Disponível em http://java.sun.com/products/jfc/tsc/articles/threads/threads1.html.

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

[PAG03] A. Popovici, G. Alonso, and T. Gross, *Just-in-time aspects: efficient dynamic weaving for java*, Proceedings of the 2nd International Conference on Aspect-oriented Software Development, 2003, pp. 100–109.

[PGA02] Andrei Popovici, Thomas Gross, and Gustavo Alonso, *Dynamic weaving for aspect-oriented programming*, Proceedings of the 1st International Conference on Aspect-Oriented Software Development, Abril 2002.

[SCT03] Yoshiki Sato, Shigeru Chiba, and Michiaki Tatsubori, *A selective, just-in-time aspect weaver*, Lecture Notes in Computer Science, Springer-Verlag GmbH, 2003, pp. 189 – 208.

[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).

[Tri98] Mario Triola, *Introdução à Estatística*, LTC, 1998, Tradução da 7ª edição.

[Vas04] Alexandre Vasseur, *Dynamic AOP and Runtime Weaving for Java – How does AspectWerkz Address It*, AOSD'04, Dynamic AOP Workshop (2004).

[VGH+00] Raja Vallée-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.

[VHS+99] Raja Vallée-Rai, Laurie Hendren, Vijay Sundaresan, Patrick Lam, Etienne Gagnon, and Phong Co, *Soot - a Java Optimization Framework*, Proceedings of CASCON 1999, 1999, pp. 125–135.

Livros Grátis

(http://www.livrosgratis.com.br)

Milhares de Livros para Download:

Baixar livros de Literatura
Baixar livros de Literatura de Cordel
Baixar livros de Literatura Infantil
Baixar livros de Matemática
Baixar livros de Medicina
Baixar livros de Medicina Veterinária
Baixar livros de Meio Ambiente
Baixar livros de Meteorologia
Baixar Monografias e TCC
Baixar livros Multidisciplinar
Baixar livros de Música
Baixar livros de Psicologia
Baixar livros de Química
Baixar livros de Saúde Coletiva
Baixar livros de Serviço Social
Baixar livros de Sociologia
Baixar livros de Teologia
Baixar livros de Trabalho
Baixar livros de Turismo