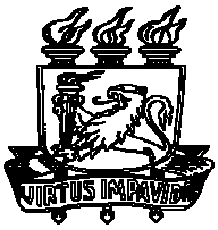

Universidade Federal de Pernambuco
Centro de Informática

Pós-graduação em Ciência da Computação

ASPECTH: UMA EXTENSÃO DE HASKELL ORIENTADA A ASPECTOS

Carlos Andreazza Rego Andrade

DISSERTAÇÃO DE MESTRADO

Recife
23 de fevereiro de 2005

Dissertação de Mestrado apresentada por ***Carlos Andreazza Rego Andrade*** a Pós-Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco, sob o título **"*Aspecth: Uma extensão orientada a aspectos de Haskell*"**, elaborada sob a orientação do **Prof. André Luis de Medeiros Santos** e aprovada pela Banca Examinadora formada pelos professores:

Prof. Paulo Henrique Monteiro Borba
(Deptº de Sistemas da Computação-CIn/UFPE)

Prof. Ricardo Massa Ferreira Lima
(Escola Politécnica/UPE)

Prof. André Luis de Medeiros Santos
(Deptº de Sistemas da Computação-CIn/UFPE)

Visto e permitida a impressão.
Recife, 23 de fevereiro de 2005.

Prof. JAELSON FREIRE BRELAZ DE CASTRO
Coordenador da Pós-Graduação em Ciência da Computação do
Centro de Informática da Universidade Federal de Pernambuco.

Universidade Federal de Pernambuco
Centro de Informática

Carlos Andreazza Rego Andrade

ASPECTH: UMA EXTENSÃO DE HASKELL ORIENTADA A ASPECTOS

Trabalho apresentado ao Programa de Pós-graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obenção do grau de Mestre em Ciência da Computação.

Orientador: *André Luís Medeiros Santos*

Recife
23 de fevereiro de 2005

Dedico este trabalho a memória de meu pai.

AGRADECIMENTOS

A despeito do empenho e dedicação que mantive durante estes quase dois anos, seria impossível concluir esta pesquisa sem o valioso auxílio de todos que direta ou indiretamente contribuíram para o desenvolvimento do trabalho:

- Primeiramente agradeço a André Santos, idealizador do projeto, cuja competente orientação e incentivo permitiram o direcionamento correto e a motivação necessária nos momentos de crise;
- A Martin Erwig, Paulo Borba e Wolfgang de Meuter, pelas discussões e comentários que esclareceram muitos pontos obscuros e certamente contribuíram para aprimorar este trabalho;
- Aos membros da banca de avaliação, Paulo Borba e Ricardo Massa, pelos comentários e sugestões importantes para melhoria deste trabalho;
- A Francisco Vieira e Augusto Sampaio, pela orientação e encorajamento;
- A Gilson Feitosa e a SEPLAN-PI, pelo apoio técnico indispensável para implementação do programa descrito no Capítulo 5;
- Aos amigos da Fábrica de Software III, Breno Costa, Carlos Júnior, Clarissa Borba, Eduardo Almeida, Fábio Ferreira, Luiz Leite, Mônica França, Taciana Vanderley, Vanessa Moura e Vivianne Medeiros, pela amizade e torcida;
- Ao Centro de Informática e ao CNPQ, pelo suporte financeiro e material necessário a realização deste trabalho;
- A minha família, pelo apoio e incentivo na busca deste objetivo;
- A Núbia Rocha, pelo carinho e compreensão durante todos os momentos;
- A Deus, pela vida.

Nada tenho, nada deixarei.

—RUBENS CUBEIRO RODRIGUES

RESUMO

Uma das principais técnicas de abstração oferecida pelas linguagens de programação atuais é a possibilidade de dividir um sistema em unidades de código que capturam suas funcionalidades. Esta abstração permite que mudanças em uma unidade em particular não se propaguem por todo sistema. No entanto, isto é apenas aplicável quando tais funcionalidades ou preocupações podem de fato ser classificadas como unidades separadas. Algumas funcionalidades ou preocupações - conhecidas como preocupações entrelaçadas (*crosscutting concerns*) - repercutem por todo o sistema e não podem ser definidas em módulos tradicionais. Assim, todo o código que as implementa fica espalhado e misturado por diversos módulos.

O paradigma de programação orientado a aspectos (AOP) tem sido apresentado na literatura como uma maneira alternativa de implementar os *crosscutting concerns* de um sistema, disponibilizando construções que permitem separá-los em unidades adequadamente. Um conceito central neste paradigma é o de aspecto, que é sua unidade modular. Uma linguagem de programação orientada a aspectos é usualmente estabelecida como uma extensão de uma linguagem de programação existente, provendo ao programador tanto as unidades modulares desta linguagem (conhecida como linguagem base) quanto os aspectos.

Este trabalho apresenta a linguagem AspectH, uma extensão orientada a aspectos de Haskell. AspectH implementa o mecanismo AOP de *pointcuts* e *advice* como em AspectJ e foi projetada para atuar em programas Haskell que utilizam mônadas. Por meio de AspectH, investigamos os benefícios que uma abordagem AOP pode oferecer no contexto de uma linguagem puramente funcional. Em outras palavras, pretendemos demonstrar que AOP pode fazer por Haskell e linguagens funcionais o que já faz, como exemplo, por linguagens orientadas a objetos.

AspectH oferece ao programador a possibilidade de implementar os *crosscutting concerns* de programas monádicos em aspectos, ajudando-o a criar programas mais modulares e conseqüentemente mais legíveis, mais fáceis de manter e reusar.

Palavras-chave: Programação orientada a aspectos, desenvolvimento de software orientado a aspectos, separação de preocupações, linguagens de programação, programação funcional, Haskell, mônadas.

ABSTRACT

One of the major abstration techniques provided by programming languages is the possibility to divide a system in code units that capture separate concerns of the system. This allows that changes made in a particular unit do not propagate through the entire system. However, this is only applicable when these concerns can indeed be classified as separate units. Some concerns – known as *crosscutting concerns* – are system-wide and can not be defined in a module. Hence all the code that deals with these concerns remains scattered an tangled in several modules.

The aspect-oriented programming paradigm (AOP) has been presented in the literature as a alternative to implement such concerns, providing abstractions that allow separation of these concerns in units. A central concept in this paradigm is the aspect, that is its modular unit. An aspect-oriented programming language is often estabilished as an extension of an existing programming language, providing to the programmer the modular units of this language (known as base language) as well as the aspects.

This work presents AspectH, an aspect-oriented extension of Haskell. AspectH implements AOP through pointcuts and advice as in AspectJ and was designed to be used in Haskell programs that use monads. Through AspectH, we evaluate the benefits that an AOP approach can offer in the context of a purely functional language. In other words, we intend to show that AOP can make for Haskell and functional programming what it already does for object-oriented languages.

AspectH offers to a programmer the possibility of implementing the crosscutting concerns of monadic programs using aspects, helping her to create modular programs and consequently more readable, easier to mantain and reuse.

Keywords: Aspect-oriented programming, aspect-oriented software development, separation of concerns, programming languages, functional programming, Haskell, monads.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

CAPÍTULO 1

INTRODUÇÃO

Uma das principais técnicas de abstração oferecida pelas linguagens de programação atuais é a possibilidade de dividir um sistema em unidades de código que capturam suas funcionalidades. Tais unidades de código incluem funções, procedimentos, métodos e classes. Esta abstração permite que mudanças em uma unidade em particular não se propaguem por todo sistema. No entanto, isto é apenas aplicável quando tais funcionalidades ou preocupações podem de fato ser classificadas como unidades separadas. Algumas funcionalidades ou preocupações – conhecidas como preocupações entrelaçadas (*crosscutting concerns*) – repercutem por todo o sistema e não podem ser definidas em módulos tradicionais. Assim, todo o código que as implementa fica espalhado e misturado por diversos módulos.

O paradigma de programação orientado a aspectos (AOP) [KLM+97] tem sido apresentado na literatura como uma maneira alternativa de implementar tais preocupações, disponibilizando abstrações que permitem separá-las em unidades adequadamente. A literatura apresenta diversos relatos de instâncias de AOP implementadas utilizando abordagens distintas e que já foram catalogadas e classificadas [MK03] em termos de suas propriedades características.

Um conceito central neste paradigma é o de aspecto, que é sua unidade modular. Uma linguagem de programação orientada a aspectos é usualmente estabelecida como uma extensão de uma linguagem de programação existente, provendo ao programador tanto as unidades modulares desta linguagem (conhecida como linguagem base) quanto os aspectos. Cabe ao programador então discernir o que deve ser implementado nos módulos da linguagem base e o que deve ser definido em aspectos.

Grande parte dos experimentos e pesquisas envolvendo AOP se concentram no paradigma orientado a objetos, por ser este o dominante atualmente. Uma implementação de destaque é AspectJ [KHH+01] que é uma extensão orientada a aspectos de Java [AG98], sendo considerada referência no tocante ao mecanismo AOP de *pointcuts* e *advice*. Esta linguagem permite a especificação de código adicional (*advice*) que executa em determinados pontos de um programa definidos pelos *pointcuts*, permitindo desta forma a criação de sistemas modulares mesmo na presença de *crosscutting concerns*.

Este trabalho apresenta a linguagem AspectH, uma extensão orientada a aspectos de Haskell. AspectH implementa o mecanismo AOP de *pointcuts* e *advice* como em AspectJ e foi projetada para atuar em programas Haskell que utilizam mônadas. Por meio de AspectH, investigamos os benefícios que uma abordagem AOP pode oferecer no contexto de uma linguagem puramente funcional. Em outras palavras, pretendemos demonstrar que AOP pode fazer por Haskell e linguagens funcionais o que já faz, como exemplo, por linguagens orientadas a objetos.

AspectH é um superconjunto de Haskell, ou seja, programas válidos em Haskell também o são em AspectH. Em AspectH, um aspecto é uma entidade similar a um

módulo convencional de Haskell onde, além de todas as declarações de Haskell, é possível a definição de declarações de *advice*. Cada declaração de *advice* possui um *pointcut* associado que especifica de maneira declarativa os pontos do programa afetados pelo *advice*. Um aspecto pode modificar código monádico tanto de módulos convencionais quanto de outros aspectos.

Mônada, um conceito importado da teoria das categorias, é utilizada em linguagens funcionais para a implementação de funcionalidades como manipulação de estado e exceções e realização de interações de I/O, que normalmente destroem a transparência referencial destas linguagens. Mônadas são ferramentas práticas importantes, uma vez que provêem um *framework* uniforme para descrever tais funcionalidades sem a necessidade de renunciar às propriedades de uma linguagem puramente funcional. Entretanto, observa-se na prática que programas monádicos, por vezes, possuem os mesmos problemas resultantes da modularização inadequada de seu código.

AspectH oferece ao programador a possibilidade de implementar os *crosscutting concerns* de programas monádicos em aspectos, ajudando-o a criar programas mais modulares e conseqüentemente mais legíveis, mais fáceis de manter e reusar.

1.1 ORGANIZAÇÃO DA DISSERTAÇÃO

Esta dissertação está organizada da seguinte forma:

- No Capítulo 2 apresentamos a Programação Orientada a Aspectos. Expomos as idéias gerais envolvidas, juntamente com os principais termos e conceitos, as vantagens e as desvantagens associadas com AOP. Descrevemos ainda as diversas tecnologias relacionadas com AOP, bem como as características que definem um mecanismo em particular como instância deste paradigma.

- No Capítulo 3 apresentamos brevemente a linguagem Haskell, com explicações e demonstrações referentes aos principais recursos disponibilizados pela linguagem, a fim de facilitar o entendimento dos programas expostos no restante da dissertação.

- No Capítulo 4 discorremos em torno da noção de mônada, oferecendo descrições a respeito de seu conceito, de como é feita a definição de uma mônada em Haskell e das facilidades providas pela linguagem à programação monádica, bem como expondo mônadas relevantes fornecidas por Haskell e freqüentemente usadas na prática. Apresentamos também os importantes tópicos a respeito de composição de mônadas e os transformadores monádicos que visam a integração de mais de uma funcionalidade simultaneamente por meio de mônadas diferentes.

- No Capítulo 5 descrevemos detalhadamente as principais características e recursos providos por AspectH. A apresentação inclui um pequeno programa que ilustra a implementação de exemplos clássicos de *crosscutting concerns* como autenticação e controle de exceções em aspectos.

- No Capítulo 6 apresentamos detalhes acerca da implementação da linguagem AspectH. A princípio apresentamos as características principais do pré-processador

de AspectH, detalhando os principais recursos utilizados, as transformações de código que implementam o processo de combinação dos aspectos com o código base e as questões de tipo relacionadas com o processo de combinação. A seguir descrevemos um estudo de caso para AspectH onde utilizamos a linguagem para (re)implementar seu próprio pré-processador separando os *crosscutting concerns* do programa base, obtendo-se mais legibilidade e modularidade no sistema resultante. O capítulo inclui uma discussão referente aos resultados obtidos no estudo de caso e à abordagem AspectH de modo geral, encerando com informações sobre a estratégia de introdução tardia e seletiva de mônadas em um programa (empregada na implementação do pré-processador de AspectH) resultando em maior grau de produtividade no desenvolvimento de programas funcionais monádicos.

- No Capítulo 7 apresentamos nossas conclusões, onde também relatamos os principais trabalhos relacionados, juntamente com os trabalhos futuros nesta linha de pesquisa.

CAPÍTULO 2

PROGRAMAÇÃO ORIENTADA A ASPECTOS

Modelar entidades de um domínio em termos de módulos em uma linguagem de programação é de fundamental importância durante o processo de desenvolvimento de um sistema. Isto porque decisões equivocadas a respeito de seus blocos conceituais eventualmente resultam em grandes dificuldades na fase de implementação, em sua manutenção e evolução, bem como criam obstáculos para um possível reuso de código em outras aplicações.

Além da complexidade inerente a este processo, existem determinados requisitos ou preocupações (normalmente relacionados a requisitos não-funcionais de uma aplicação) que se ajustam de forma inadequada ao sistema de módulos das linguagens de programação atuais. Na prática, isto significa que o código referente a estas preocupações fica *espalhado* pelos módulos de alguma forma envolvidos com a implementação de tais preocupações (*code scattering*). Desta maneira, em um determinado módulo é possível encontrar, além da funcionalidade "principal" implementada por ele, código referente a diversas outras preocupações (*código misturado – code tangling*), prejudicando a legibilidade e, conseqüentemente, aumentando a complexidade do programa.

Como exemplo de tais preocupações, podemos citar *logging* de operações no sistema e manipulação de erros e exceções: por meio dos mecanismos tradicionais (funções, métodos, classes, etc.) disponibilizados pelas linguagens de programação, é impossível implementar estas funcionalidades de forma modular. Em uma linguagem procedural, estas preocupações podem estar implementadas como partes de procedimentos disjuntos; em uma linguagem orientada a objetos, podem estar espalhadas por vários métodos ou mesmo classes.

A modularização deficiente destas preocupações – aqui por diante referidas como *preocupações entrelaçadas* (*crosscutting concerns*) – representa um impacto negativo considerável para o sistema, resultando em diversas complicações:

- **Dificuldade em localizar o código de uma funcionalidade específica**. A implementação simultânea de várias preocupações torna pouco visível a correspondência entre uma preocupação e sua implementação, resultando em um mapeamento inadequado entre as duas.

- **Reuso de código comprometido**. Tendo em vista que os *crosscutting concerns* não são unidades separáveis do sistema, é mais difícil o reuso do código que os implementa em outros programas.

- **Comprometimento da qualidade do código**. Código misturado é repositório propício a problemas camuflados, difíceis de serem identificados e solucionados. Ademais, a atenção do programador permanece voltada a diversas preocupações

simultaneamente, podendo uma ou mais destas preocupações receber atenção insuficiente.

- **Evolução dificultada**. Devido à falta de modularidade, planejar e implementar requisitos futuros (o que implica eventualmente na reimplementação de determinadas partes do sistema) resulta na modificação de muitos módulos, levando, por vezes, à inconsistências.

Este capítulo apresenta Programação Orientada a Aspectos (*Aspect Oriented Programming – AOP*), uma tecnologia dedicada a minimizar os efeitos negativos que a falta de modularidade do código referente às *preocupações entrelaçadas* podem causar a um sistema, auxiliando a separação de preocupações [Par72] no desenvolvimento do mesmo. Apresentamos os principais termos e conceitos, as etapas para o desenvolvimento utilizando AOP, os benefícios que a utilização disciplinada das técnicas providas pode oferecer durante a implementação de um sistema, bem como as desvantagens que esta metodologia pode trazer (Seção 2.1). Expomos ainda as propriedades que caracterizam um mecanismo ou linguagem de programação como instância de AOP (Seção 2.2). Por fim, na Seção 2.3, discutimos mecanismos relacionados com AOP, apresentando as principais idéias associadas a eles e ao menos uma instância de cada.

2.1 VISÃO GERAL DE PROGRAMAÇÃO ORIENTADA A ASPECTOS

Programação Orientada a Aspectos é uma metodologia de programação que provê as abstrações necessárias para expressar claramente os *crosscutting concerns* de um sistema. Neste contexto, "expressar claramente" deve ser entendido como o isolamento apropriado, a composição e o reuso de código referente a tais funcionalidades.

Temos, então, a definição de um conceito central em AOP, *aspecto*, que é sua unidade modular. Um aspecto captura a implementação de um *crosscutting concern*, encapsulando-o de maneira adequada.

Um mecanismo AOP pode ser implementado como uma extensão de algum paradigma de programação existente (como, por exemplo, procedural ou orientado a objetos), sendo este referenciado como paradigma ou linguagem base. As funcionalidades adequadamente encapsuladas através das construções disponibilizadas pela linguagem base são implementadas desta forma, enquanto que os *crosscutting concerns* são implementados em aspectos. Projetar um sistema AOP [1], portanto, envolve o entendimento do que deve ser implementado na linguagem base, o que deve ser implementado na linguagem de aspectos e o que deve ser compartilhado entre elas.

O desenvolvimento de um sistema AOP inclui um programa escrito na linguagem base, um (ou mais) programa(s) na linguagem de aspectos e um processo que componha o programa final – possivelmente em termos da linguagem base – refletindo as contribuições semânticas dos programas originais. O processo de composição é conhecido como *weaving* e fica a cargo do *aspect weaver* ou combinador, uma espécie de compilador [2].

[1] Entenda-se por "sistemas AOP" os que podem ser claramente beneficiados por um mecanismo AOP.

[2] Como veremos mais a frente neste capítulo, um mecanismo AOP não implica necessariamente na existência de uma linguagem de aspectos e de um *aspect weaver*. O aspecto pode ser implementado, por exemplo, pela aplicação sistemática de recursos providos pela própria linguagem base.

O *aspect weaver* utiliza as chamadas *regras de recomposição* ou *regras de weaving*, definidas juntamente com os aspectos, para especificar de que maneira será feita a composição dos aspectos com o programa base.

A eficácia de AOP está justamente aí: o *aspect weaver* cuida dos detalhes referentes à composição dos aspectos com o programa base que, de outra maneira seria feita manualmente pelo programador. Explicando de outra forma, o programa resultante do processo de *weaving* possui os problemas descritos no início deste capítulo; entretanto, a "mistura" e o "espalhamento" de código não fica mais sob responsabilidade do programador, não tendo este a necessidade de lidar diretamente com estes detalhes.

Linguagens AOP e o *aspect weaver* podem ser projetados de maneira que o processo de *weaving* seja protelado para ocorrer durante a execução (*weaving* dinâmico) ou para ser feito estaticamente.

A Figura 2.1 ilustra graficamente as diferenças existentes entre o processo de compilação tradicional e o utilizado por uma linguagem AOP. À esquerda temos o processo tradicional onde, a partir dos componentes de um programa, o compilador gera uma representação executável do mesmo. À direita temos um cenário possível em um mecanismo AOP: a primeira etapa é a composição dos componentes do código base e dos aspectos pelo *aspect weaver*; por fim, o programa resultante do processo de *weaving* é compilado como de costume.

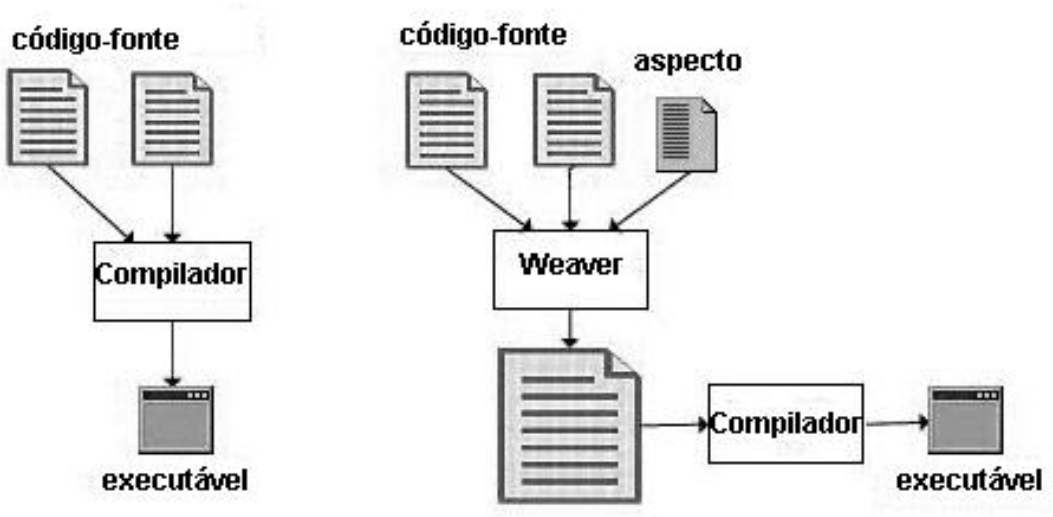

Figura 2.1. Processo de weaving

Um elemento crítico no projeto de um mecanismo AOP é o modelo de *join points*. *Join points* são elementos da semântica da linguagem base onde é permitida a influência do código dos aspectos.

O modelo de *join points* provê uma referência comum que possibilita a execução coordenada do código base e dos aspectos. *Aspect weavers* funcionam por meio da geração de uma representação de *join points* do programa base. Feita esta geração, o código dos aspectos é executado (ou compilado) em relação a esta representação.

2.1.1 Etapas do desenvolvimento AOP

AOP permite a implementação das preocupações de um sistema de maneira fracamente acoplada, combinando estas implementações de modo a constituir o sistema final. Em outras palavras, AOP produz sistemas utilizando implementações modulares de *crosscutting concerns*.

Laddad [Lad03] define três etapas para o desenvolvimento de sistemas AOP. A seguir descrevemos brevemente estas etapas.

i) **Decomposição de aspectos**. Esta etapa é responsável pela decomposição dos requisitos de um sistema com o objetivo de identificar preocupações "normais" e as entrelaçadas. Nesta fase, identifica-se o que deve ser implementado na linguagem base e o que será implementado em aspectos.

ii) **Implementação de preocupações**. Nesta etapa é feita a implementação das preocupações separadamente.

iii) **Recomposição de aspectos**. Nesta etapa o *aspect weaver*, por meio de regras de recomposição definidas nos aspectos, realiza a composição dos aspectos com o programa base, resultando no sistema final, freqüentemente em termos da linguagem base.

Laddad emprega a metáfora dos prismas – onde a luz é decomposta em suas partes constituintes ao passar por um prisma e, posteriormente, recomposta ao passar por outro – para ilustrar as etapas do processo AOP. O raio de luz representa os requisitos que são identificados ao transpor o primeiro prisma. Em seguida estes requisitos são implementados separadamente; por fim, é feita a recomposição das implementações pelo *aspect weaver*. A Figura 2.2 ilustra graficamente a metáfora utilizada.

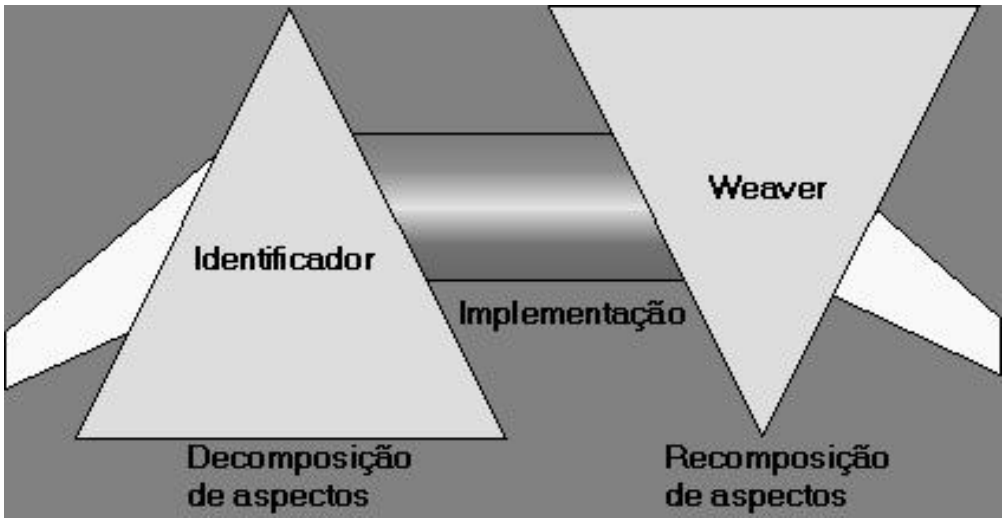

Figura 2.2. Etapas do desenvolvimento AOP. (Fonte: AspectJ in Action).

2.1.2 Benefícios

As abstrações providas por um mecanismo AOP ajudam a transpor as barreiras impostas pelo tratamento inadequado dos *crosscutting concerns*. A seguir apresentamos os principais benefícios que AOP pode proporcionar ao desenvolvimento de um programa.

- **Implementação modular de crosscutting concerns**. AOP lida com cada preocupação separadamente, com o mínimo de acoplamento, resultando em implementações modulares mesmo na presença de *crosscutting concerns*. Tal implementação origina sistemas com menos código duplicado, diminuindo-se a complexidade, com ganhos em legibilidade.

- **Sistemas mais fáceis de manter**. Uma vez que o código base está separado do código dos *crosscutting concerns*, é mais fácil adicionar-se novas funcionalidades em novos aspectos. Além disso, quando se adiciona novos módulos ao sistema, é possível que os aspectos existentes já influenciem na semântica deles, ajudando assim a criar uma evolução mais coerente.

- **Mais reuso de código**. AOP implementa cada preocupação como um módulo separado, diminuindo-se o acoplamento entre os módulos do sistema, facilitando o reuso de código em outras aplicações.

Em geral, uma implementação fracamente acoplada representa a chave para mais reuso de código.

2.1.3 Desvantagens

Até aqui, descrevemos AOP como uma metodologia de programação que proporciona diversos benefícios ao programador, oferecendo construções que aumentam a modularidade de um sistema. O leitor, contudo, não entenda AOP como uma panacéia destinada a resolver todos os males que o tratamento inadequado dos *crosscutting concerns* pode trazer a um sistema. AOP tem restrições e desvantagens.

Um inconveniente resultante de uma abordagem AOP se refere ao entendimento de um módulo em particular. Tendo em mente que a semântica de um módulo pode estar sendo influenciada por aspectos definidos remotamente, é impossível apreciar o funcionamento integral de tal módulo sem ter em vista todos os aspectos que de alguma maneira influenciam seu significado. Verificar propriedades de um programa em um cenário onde é permitido modificar arbitrariamente a semântica de qualquer módulo, por meio de código advindo de locais completamente não relacionados, pode se tornar uma tarefa bastante complexa. É importante que as construções poderosas oferecidas por uma metodologia AOP sejam empregadas de maneira disciplinada (como, evidentemente, é o recomendado para qualquer abstração disponibilizada por uma metodologia de programação).

Estas observações, vistas de uma ótica puramente pragmática, podem ser apreendidas como ortodoxia excessiva. Entretanto, considerando a cultura – programação funcional e, em particular, Haskell – em que pretendemos conduzir a tecnologia discutida neste

capítulo, esta discussão toma uma dimensão mais ampla, sendo até indispensável. Isto porque a atividade de realização de provas de propriedades de programas é bem mais freqüente em linguagens cujas expressões preservam a transparência referencial (o que é o caso de Haskell).

Outra questão importante a ser considerada é a composição propriamente dita de aspectos com o código base. Em um contexto ideal teríamos programadores responsáveis pela implementação da funcionalidade básica de um sistema e outros responsáveis pela implementação dos *crosscutting concerns* na linguagem de aspectos, sendo aqueles inconscientes do trabalho empreendido por estes. Sistemas compostos por mentes distintas, por meio de regras de composição formais podem não funcionar da maneira esperada, podendo apresentar resultados inesperados [Fil01].

2.2 PROPRIEDADES CARACTERÍSTICAS DE AOP

Existem diversas tecnologias apresentadas na literatura como instâncias de AOP. Algumas são implementadas utilizando-se técnicas como padrões de projetos [GHJV94], ou por meio da aplicação sistemática de recursos providos pela própria linguagem base (como, por exemplo, introspecção). Outras envolvem a existência de uma (ou mais) linguagem(ns) para especificação de aspectos e regras de recomposição.

Em vista disso, um questionamento importante emerge: o que caracteriza um mecanismo de programação como sendo instância de AOP, isto é, o que o torna apropriado à implementação de sistemas AOP?

Filman e Friedman [FF00, Fil01] sustentam que são necessárias duas propriedades para definir um mecanismo como instância de AOP: *quantificação* e "*inconsciência*" (*obliviousness*).

2.2.1 Quantificação

Quantificação é a idéia de que é possível escrever declarações unitárias e individuais que têm influência em muitos pontos em um programa. Em outras palavras, é permitido definir em um sistema AOP o código que será executado quando determinadas circunstâncias ocorrem no sistema. Há duas formas de quantificação: *estática* e *dinâmica*.

Quantificação estática é realizada acerca da representação textual do programa. Tal representação pode ser entendida em termos das interfaces públicas do programa, ou como a árvore resultante da análise sintática do mesmo (sintaxe abstrata). Estas duas interpretações distintas levam à seguinte classificação de sistemas AOP:

- **AOP caixa-preta**. Sistemas AOP caixa-preta quantificam relativamente à interface pública de componentes como funções e métodos de objetos. Tais sistemas podem ser implementados sem a necessidade do código-fonte.

- **AOP caixa-branca**. Sistemas AOP caixa-branca permitem a quantificação sobre a sintaxe abstrata dos componentes de um programa. Tais sistemas permitem acesso a todas as sutilezas (estáticas) do mesmo. Nestes sistemas o código-fonte do programa é requerido.

Quantificação dinâmica é realizada como uma resposta a um evento que ocorre durante a execução do programa. Exemplos de tais eventos incluem: levantamento de exceções, chamada de uma rotina dentro do escopo temporal de uma chamada a uma outra rotina e padrões de execução de um programa (por exemplo, após a execução da rotina "verificar senha" falhar cinco vezes seguidas).

2.2.2 Obliviousness

É possível imaginar um cenário onde se tenha programadores dedicados à implementação da funcionalidade básica de um sistema e outros destinados ao código dos aspectos. Para que um mecanismo seja considerado essencialmente instância de AOP é requerido que os programadores do primeiro grupo sejam inconscientes do código dos aspectos, isto é, que não precisem despender qualquer esforço adicional para fazer o mecanismo AOP funcionar. Tal propriedade é conhecida como *obliviousness*.

Obliviousness declara que é impossível discernir que o código definido em aspectos executará tão-somente examinando o código do programa base. Esta propriedade é desejável porque permite uma maior separação de preocupações durante o processo de criação de um sistema.

2.3 MECANISMOS DE PROGRAMAÇÃO ORIENTADA A ASPECTOS

Nesta seção discutimos cinco mecanismos reconhecidos na literatura como instâncias de AOP. A nomenclatura dos mecanismos utilizada aqui, com exceção de "navegadores baseados em consultas", é derivada dos trabalhos de Masuhara e Kiczales descritos em [MK03], onde os autores apresentam um *framework* para avaliar mecanismos AOP.

Os mecanismos descritos nas próximas subseções são:

- *Pointcuts* e *advice*;
- especificação de travessias;
- composição de hierarquias de classes;
- classes "abertas" e
- navegadores baseados em consultas.

Para cada mecanismo apresentamos as idéias básicas e pelo menos uma instância do mesmo, com informações a respeito de sua implementação e utilização.

2.3.1 Pointcuts e advice

O que caracteriza o modelo de *pointcuts* e *advice*, como implementado, por exemplo, em AspectJ [KHH+01] e AspectC [CKFS01], é a utilização de *advice* para modificar incrementalmente o comportamento de um programa. A declaração de *advice* especifica uma ação que será executada sempre que uma determinada condição é atendida durante a execução de um programa. Tais condições estão diretamente relacionadas com o modelo

de *join points*, definido pelo mecanismo AOP. A execução do *advice* correspondente em um *join point* fica a cargo do *aspect weaver*.

A identificação de *join points* em um programa é feita através de *pointcuts*, que são predicados que descrevem os *join points* onde se deseja especificar uma computação adicional (*advice*). Um *pointcut* pode ser visto, então, como o conjunto de *join points* que satisfazem o predicado especificado por ele.

Pointcuts e *advice* não são entidades estritamente independentes. O *pointcut* pode disponibilizar informações de contexto em um *join point* que podem ser utilizadas no escopo da computação definida pelo *advice*. Tais informações podem ser, por exemplo, os argumentos da chamada de uma função. Cada *advice* possui a declaração de um *pointcut* que especifica os *join points* que serão modificados por ele. As modificações podem tão-somente especificar uma computação adicional que será executada no *join point*, como substituir toda a computação definida no *join point* por uma outra.

Poincuts e *advice* são, desta forma, a maneira utilizada pelas linguagens que os implementam para fazer declarações quantificadas acerca de um programa. A linguagem definida para expressar um *pointcut* deve ser poderosa o suficiente para identificar os *join points* de forma adequada.

Para ilustrar os conceitos de *pointcut* e *advice*, examinemos como eles foram implementados em AspectJ. AspectJ é uma linguagem de propósito geral que estende Java com construções que permitem a implementação modular de sistemas AOP. O compilador (*aspect weaver*) de AspectJ produz classes que satisfazem a especificação de *byte code* [3] Java, podendo desta forma executar em qualquer máquina virtual Java.

AspectJ permite a implementação de duas espécies de *crosscutting concerns*, uma das quais é baseada em *pointcuts* e *advice* (discutida aqui); a outra permite a definição de novas operações em classes existentes (discutida na Seção 2.3.4).

Em AspectJ, um aspecto é definido de forma similar à declaração de uma classe em Java. A declaração de aspecto pode incluir declarações de *pointcuts* e *advice*, bem como todas as outras declarações permitidas na definição de uma classe.

Um *pointcut* em AspectJ pode selecionar, entre outros *join points*, a chamada de um método ou construtor, a execução de um método ou construtor, a leitura e a modificação de um atributo e a execução de um manipulador de exceções. *Pointcuts* em AspectJ não são de alta ordem, nem paramétricos.

A especificação de *pointcuts* é feita por meio de operadores primitivos que descrevem os *join points* que se deseja selecionar. Exemplos de tais operadores são *call* (intercepta a chamada de um método ou construtor), *execution* (execução de um método ou construtor), *get* (leitura de um atributo), *set* (escrita em um atributo) e *within* (qualquer *join point* dentro do escopo léxico de uma classe).

Como exemplo, o *pointcut* a seguir:

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

seleciona todas as chamadas ao método *setX* da classe *Point*, que possui um parâmetro formal do tipo *int* e cujo tipo de retorno é *void*.

[3] Código intermediário utilizado por máquinas virtuais Java.

É permitida a definição de *pointcuts* mais elaborados por meio do operador unário !(negação) e dos operadores binários ||(disjunção) e &&(conjunção). O *pointcut* composto a seguir seleciona as chamadas aos métodos interceptados pelos operandos de ||:

```
call (void Point.setX(int)) ||
call (void Point.setY(int))
```

Pointcuts em AspectJ podem ser definidos de duas maneiras: de forma anônima, ou através da primitiva **pointcut**, que associa a definição de um *pointcut* a um identificador. *Pointcuts* anônimos são definidos no local de sua utilização, tal como parte constituinte de um *advice* ou na definição de outro *pointcut*.

A declaração a seguir especifica um *pointcut* cuja definição está ligada ao identificador *moves*:

```
pointcut moves():
  call (void Point.setX(int)) ||
  call (void Point.setY(int))
```

Advice em AspectJ é uma entidade similar a um método em Java. O código definido em um *advice* executa em cada *join point* selecionado pelo *pointcut* associado a ele. Há três tipos de *advice* em AspectJ, a saber, *before,* after e *around*. Existem, ainda, duas variações de *advice after* que são *after returning* e *after throwing*, correspondentes às duas formas que uma computação pode retornar de um *join point* (de forma normal ou disparando uma exceção). O código de um *advice before* executa antes da computação definida em um *join point*, enquanto que, no caso de um *advice after*, tal execução é feita após a computação do *join point*. O *advice around* executa seu código em substituição ao definido em um *join point* podendo, contudo, restabelecer a computação do *join point* através de uma chamada ao (pseudo)método *proceed*.

Para exemplificar, observemos o *advice* seguinte:

```
after(): moves(){
  flag = true;
}
```

A definição estabelece que o comando *flag = true* executa sempre que um *join point* no programa atenda ao predicado definido no *pointcut moves*.

Vejamos agora a definição completa de um aspecto em AspectJ que utiliza as declarações analisadas nesta subseção. O aspecto é utilizado para detectar alterações nas coordenadas de um objeto da classe *Point*:

```
1. aspect MoveTracking {
2.
3.   static boolean flag = false;
4.
5.   pointcut moves():
6.     call (void Point.setX(int)) ||
```

```
7.      call (void Point.setY(int));
8.
9.    after(): moves() {
10.     flag = true;
11.   }
12. }
```

Em AspectJ a definição de *advice* possui acesso a todas os membros definidos no aspecto. Neste caso, o advice referencia a variável estática *flag* (na linha 10).

2.3.2 Especificação de travessias

Uma operação em um programa orientado a objetos freqüentemente envolve várias classes diferentes que colaboram entre si. Usualmente, existem duas formas para implementar tal operação: ou se coloca toda a operação em um único método de uma destas classes, ou divide-se a implementação da operação em cada uma das classes envolvidas. A desvantagem da primeira abordagem é que muita informação relativa à estrutura das classes (relacionamentos de herança e composição) fica misturada no método que implementa a operação, dificultando a adaptação quando ocorrem mudanças em tal estrutura. O inconveniente da segunda abordagem é que o código da operação fica espalhado por diversas classes, dificultando a adaptação quando ocorrem mudanças na operação.

Para resolver este dilema *Programação Adaptativa* [Lie96] define a noção de *método adaptativo*. Um método adaptativo encapsula o comportamento de uma operação, evitando desta forma o problema de espalhamento de código. Ele também impede os transtornos causados pela mistura de código, já que a noção também abstrai com relação a estrutura das classes.

A idéia de método adaptativo é especificar travessias pela estrutura das classes em alto nível. Uma travessia é um caminhamento através de um grupo de objetos relacionados com o propósito de realizar uma operação.

Na especificação da travessia, o comportamento é expresso por meio da descrição de como alcançar os participantes da computação, juntamente com a operação a ser realizada quando cada participante é atingido (expressa através de um *Visitor* [GHJV94]). Tanto a especificação da travessia quanto o *Visitor* mencionam apenas um conjunto mínimo de classes envolvidas na operação; a informação a respeito das conexões entre as classes é abstraída.

No contexto de AOP, é possível entender a especificação da travessia como uma maneira de implementar de forma modular os *crosscutting concerns* relativos ao comportamento de uma hierarquia de classes. As unidades modulares (aspectos) seriam, então, os métodos adaptativos que encapsulam as informações relacionadas ao comportamento das classes. Os *join points* são definidos indiretamente pela estratégia de travessia, enquanto que o *Visitor* corresponde ao *advice*.

DemeterJ [LO97] e DJ [OL01, LOO01] implementam a estratégia de especificação de travessias empregando abordagens distintas.

DemeterJ realiza as especificações de travessias por meio de uma pequena linguagem. O compilador DemeterJ utiliza como entrada um arquivo com informações a respeito

da estrutura das classes (dicionário de classes) e um conjunto de arquivos que contêm código Java, estratégias de travessia e métodos do *Visitor*. Em seguida, o compilador gera código Java que realiza o caminhamento na estrutura de classes do programa, fazendo neste trajeto chamadas aos métodos do *Visitor*.

Vejamos a seguir um exemplo de especificação de travessias e de um *Visitor* em DemeterJ:

```
1. Conglomerate {
2.   traversal allSalaries(SummingVisitor s) {
3.     to Salary
4.   }
5. }
6.
7. SummingVisitor {
8.   before Salary (@ total = total.add(host.get_v()); @)
9. }
```

O código define o caminhamento *allSalaries* (linhas 2 a 4) por todos os objetos da classe *Salary* (*to Salary*, linha 3). Em cada objeto alcançado pela travessia é feita a execução do código do *SummingVisitor*, que computa a soma dos valores armazenados nos objetos da classe *Salary*. Para anexar outro *Visitor* à travessia basta declarar o mesmo como parâmetro de *allSalaries*.

DJ é uma biblioteca que utiliza os recursos de reflexão de Java [Sun96] para especificar as travessias. Pelo fato de utilizar reflexão, torna-se desnecessária a existência de um pré-processador como é o caso em DemeterJ. As travessias são expressas por meio de *Strings* em um código comum Java.

Analisemos o código abaixo que exemplifica um método adaptativo implementado em Java usando a biblioteca DJ. O propósito do código é somar os valores armazenados em objetos da classe *Salary*.

```
1.  import edu.neu.ccs.demeter.dj.ClassGraph;
2.  import edu.neu.ccs.demeter.dj.Visitor;
3.
4.  class Company {
5.    static ClassGraph cg = new ClassGraph(); // estrutura de classes
6.    Double sumSalaries() {
7.      String s = "from Company to Salary";   // travessia
8.      Visitor v = new Visitor() {            // Visitor
9.        private double sum;
10.       public void start() { sum = 0.0 };
11.       public void before(Salary host) { sum += host.getValue(); }
12.       public Object getReturnValue() { return new Double(sum); }
13.     };
14.     return (Double) cg.traverse(this, s, v);
15.   }
```

```
16.  (...)
17. }
```

A travessia é definida por meio da *String "from Company to Salary"* (linha 7). O objeto da classe *Visitor* (linhas 8 a 13) especifica o código a ser executado em cada objeto da classe *Salary* alcançado pela travessia. A chamada ao método *traverse* da classe *ClassGraph* (linha 14) realiza a travessia propriamente dita, especificando como argumento o objeto a partir do qual deve iniciar o caminhamento (*Company*), a travessia *s* e o *Visitor v*.

Uma desvantagem de especificar a travessia por meio de reflexão Java é o impacto negativo no desempenho do programa durante a sua execução.

2.3.3 Composição de hierarquias de classes

Formalismos modernos para desenvolvimento de software (linguagens de programação e de modelagem, por exemplo) tipicamente permitem a decomposição de sistemas em módulos de acordo com uma única "dimensão de preocupações", conhecida como *dimensão dominante*. O formalismo freqüentemente define de forma específica o que a dimensão dominante deve ser. Mesmo quando não é imposta uma dimensão dominante (no caso de abordagens híbridas), o formalismo não suporta a decomposição simultânea de acordo com dimensões múltiplas, o que leva o desenvolvedor a definir qual dimensão será usada.

Em [TOHS99], é proposto um modelo que permite a decomposição e composição de sistemas em várias dimensões simultaneamente (*multi-dimensional separation of concerns*). Dimensões possíveis incluem dados (ou objetos), funcionalidade e funções [4].

O modelo introduz o conceito de *hyperslice* como sendo um conjunto de módulos convencionais, escritos em qualquer formalismo. *Hyperslices* encapsulam preocupações em dimensões não restringidas à dominante. Os módulos dentro de um *hyperslice* são os definidos pelo próprio formalismo, exceto que eles contêm apenas unidades pertencentes a uma única preocupação. Isto significa que estes módulos podem não satisfazer os requisitos de integridade que o formalismo normalmente impõe. Por exemplo, considerando que o módulo utilizado seja uma classe e o formalismo a linguagem Java, ter-se-ia unidades (métodos e atributos) de uma mesma classe definidos nos diversos *hyperslices* em que o sistema estiver dividido.

Neste modelo um sistema é definido como uma coleção de *hyperslices*, por meio dos quais é possível separar as preocupações de importância em um sistema, empregando-se quantas dimensões forem necessárias. A composição dos *hyperslices* define o sistema final.

A composição de um conjunto de *hyperslices* é especificada através de regras, definindo-se então um *hypermodule*, este sim, satisfazendo os requisitos de completude exigidos pelo formalismo.

HyperJ [LMW00] suporta o modelo de separação de preocupações em múltiplas dimensões para Java. A linguagem permite que um conjunto de classes seja decomposto

[4] Nesta subseção nos concentraremos especificamente na dimensão de dados e, em particular, na composição de hierarquias de classes.

em várias dimensões simultaneamente. Por exemplo, classes Java podem ser consideradas dentro da dimensão *Classe*, ao mesmo tempo que determinados métodos de tais classes podem ser considerados operações na dimensão *Funcionalidade*. Cada dimensão pode ser particionada em um conjunto de preocupações.

Para utilizar Hyper/J é necessário que o desenvolvedor forneça três arquivos:

- **Arquivo hyperspace**, que descreve as classes do programa que podem ser manipuladas por Hyper/J. A seguir apresentamos um exemplo deste arquivo que declara que todas as classes do pacote *Test* fazem parte do processo de Hyper/J:

```
hyperspace Test
   composable class Test.*;
```

- **Arquivo de mapeamento de preocupações**, que descreve os relacionamentos entre o código Java e *hyperslices*. O exemplo a seguir, relaciona a classe A com o *hyperslice Feature.Kernel* e a classe B com o *hyperslice Feature.New*:

```
class A : Feature.Kernel
class B : Feature.New
```

- **Arquivo hypermodule**, que especifica os *hyperslices* que são utilizados pelo sistema, juntamente com as regras de composição entre eles. Para exemplificar, temos a seguir um arquivo *hypermodule* que emprega os *hyperslices Feature.Kernel* e *Feature.New* e como é feita a composição entre eles; neste caso a composição será feita baseada em nomes (*mergeByName*), onde, por meio da cláusula *equate*, a classe A corresponderá à classe B:

```
hypermodule Test
   hyperslices:
      Feature.Kernel;
      Feature.New;
   relationships:
      mergeByName;
      equate class Feature.Kernel.A,
                   Feature.New.B;
```

2.3.4 Classes abertas

O mecanismo de classes abertas torna possível a definição de métodos ou atributos de classes fora da declaração textual das mesmas. Em geral, tais mudanças estáticas nas classes não afetam diretamente o comportamento delas.

O modelo de classes abertas é implementado por AspectJ. Em AspectJ, é possível declarar membros (atributos, métodos e construtores) pertencentes a classes do sistema em aspectos. Além disso, pode-se declarar que determinadas classes implementam novas interfaces ou estendem novas classes.

Como exemplo, consideremos a necessidade de se criar uma cópia idêntica de um objeto em um programa Java. Para tanto, é necessário que o objeto candidato a replicação seja subtipo da interface *Cloneable* e que implemente o método *clone*. É possível definir tais declarações em um aspecto em vez de modificar a classe em questão para conseguir o resultado desejado.

O aspecto a seguir possui duas declarações: a primeira especifica que a classe *Point* será subclasse de *Cloneable* (linha 2), enquanto que a segunda (linhas 4 a 6) declara o método *clone*, pertencente a classe *Point*:

```
1. aspect CloneablePoint {
2.   declare parents: Point implements Cloneable;
3.
4.   Object Point.clone() {
5.     return super.clone();
6.   }
7. }
```

A rigor a classe *Point* não é modificada; todavia, outras classes que fazem uso dela podem utilizar a nova "funcionalidade" definida no aspecto *CloneablePoint*.

2.3.5 Navegadores baseados em consultas

Em um ambiente de desenvolvimento de software, programadores podem ter a sua disposição uma ampla variedade de ferramentas para exploração, visualização e navegação que ajudam em tarefas relacionadas à identificação de código que realiza determinada funcionalidade. Tais ferramentas são de grande utilidade já que tarefas deste calibre podem tornar-se bastante complicadas se a funcionalidade desejada é um *cross-cutting concern* implementado de forma não modular em um programa.

Estas ferramentas abrangem navegadores que exploram hierarquias de classes, ou o grafo de chamadas de métodos em um sistema, além de uma variedade de engenhos de buscas e linguagens de consultas específicas.

JQuery [JV03] é uma ferramenta de navegação de código-fonte que permite a especificação de consultas através de uma linguagem lógica. Desta forma, a ferramenta possui as vantagens de um navegador que explora a estrutura hierárquica de um programa e as de linguagens de consultas.

Em JQuery, o usuário especifica as propriedades a serem extraídas em termos de predicados lógicos relativos à estrutura do código. Uma especificação adicional é responsável pelo agrupamento dos resultados da consulta.

Considere a consulta a seguir:

```
class(?C), re_name(?M,/^get/), method(?C,?M), modifier(?M, public)
```

que retorna pares contendo classes e métodos. Cada par é representado por um ambiente que liga as variáveis C e M aos valores do par. Os pares selecionados pela consulta são constituídos de classes e de seus métodos públicos cujos nomes iniciam com a seqüência "get".

A Figura 2.3 ilustra o resultado da execução da consulta acima, apresentado como uma representação em forma de árvore, onde cada nó contém classes com seus respectivos métodos. O agrupamento dos resultados na árvore é especificado através de "*Order of Variables*", que neste caso exibe os nomes dos métodos abaixo da classe correspondente. Caso se inverta a ordem das variáveis, obter-se-á uma representação onde as classes ficam subordinadas aos métodos.

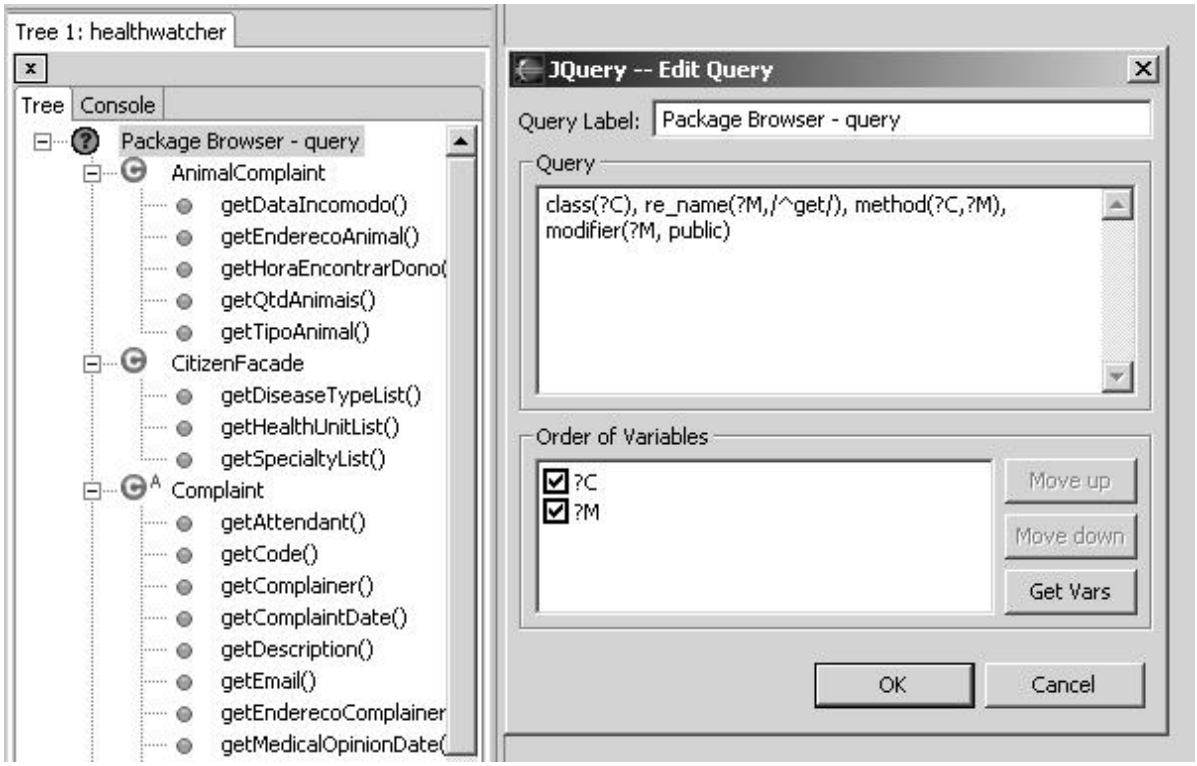

Figura 2.3. Visão hierárquica em JQuery.

2.3.6 Conclusões

Os mecanismos apresentados nas subseções anteriores representam abordagens distintas para implementação dos conceitos de AOP; entretanto, possuem o mesmo objetivo que é a descrição modular e a composição dos *crosscutting concerns* de um sistema.

No caso específico de navegadores baseados em consultas, que são ferramentas de exploração de código, tal descrição é feita em termos das árvores hierárquicas apresentadas como resultado das consultas submetidas. Não temos aqui a separação propriamente dita de código, fornecida pelos demais mecanismos. Tais ferramentas são de fundamental importância quando se deseja reestrururar um programa que apresenta deficiências quanto

a modularidade de seus *crosscutting concerns*, facilitando a definição propriamente dita do código referente a eles, que pode ser definido separadamente em aspectos.

CAPÍTULO 3

HASKELL

Programação funcional é caracterizada pelo uso de funções. Em uma linguagem funcional, um programa é uma função (ou grupo de funções), tipicamente composta de funções mais simples. Os relacionamentos entre funções são diretos: a definição de uma função pode incluir chamadas a funções, o resultado de uma função pode ser usado como argumento em chamadas de outras funções, além de ser permitida a composição entre funções.

Em uma linguagem funcional não existe a noção de estado, onde é possível criar variáveis livremente e modificar seus valores por meio de comandos de atribuição. Generalizando, programas funcionais não possuem efeitos colaterais, isto é, o único efeito permitido em uma função é calcular seu resultado. Uma conseqüência importante disto é que identificadores podem ser substituídos livremente por seus valores, uma vez que os valores armazenados não podem ser modificados pelo programa. Esta propriedade importante é conhecida como *transparência referencial*.

Uma forte motivação para o estudo de linguagens funcionais reside no fato de elas serem adequadas a técnicas tais como transformação e prova de propriedades em programas. Programas funcionais são mais fáceis de se raciocinar e transformar porque uma regra básica de prova ou transformação é mais fácil de entender e aplicar, devido à preservação da transparência referencial [Wat90].

Recursos importantes freqüentemente encontrados em linguagens funcionais incluem:

- **Avaliação preguiçosa**, que é um mecanismo de avaliação de parâmetros onde o argumento de uma função é avaliado apenas quando ele é utilizado, ao invés de se fazê-lo durante a ativação da função. Caso o argumento não seja utilizado, tampouco será avaliado, possibilitando ganhos em desempenho.

- **Funções de alta ordem**, que são funções que podem ter outras funções como argumento ou retornar uma função como resultado. Funções de alta-ordem são utilizadas para tornar o código mais reusável, abstraindo as partes do código que são específicas da aplicação. Além disso, é possível utilizar funções em estruturas de dados.

- **Polimorfismo paramétrico**, que permite a definição de abstrações que atuam em famílias inteiras de tipos de maneira uniforme. Funções polimórficas são definidas de forma genérica, ao invés de se criar uma versão diferente para cada tipo associado.

Este capítulo apresenta resumidamente a linguagem Haskell, que atualmente concentra a maior parte das pesquisas em linguagens funcionais com mecanismo de avaliação preguiçosa. Descrevemos brevemente a linguagem, explicitando suas principais características, juntamente com demonstrações dos principais recursos providos por ela.

3.1 VISÃO GERAL DE HASKELL

A linguagem Haskell foi concebida em 1987 em resposta à necessidade de uma linguagem padrão para o desenvolvimento de pesquisa no campo de programação funcional. Na época acreditava-se que a ampla variedade de linguagens disponíveis estava sufocando o desenvolvimento na área. Existiam várias linguagens puramente funcionais, nenhuma das quais, entretanto, com suporte extensivo pela comunidade de desenvolvimento e pesquisa. Atualmente, Haskell é amplamente utilizada dentro da comunidade de programação funcional, estando disponíveis diversas implementações.

O nome da linguagem é uma homenagem a Haskell B. Curry que foi um dos pioneiros do λ-cálculo (lambda cálculo), que é uma teoria matemática de funções utilizada pelas linguagens funcionais.

Haskell tem um papel preponderante no processo de popularização e no desenvolvimento de pesquisas que visam a implementação eficiente de linguagens funcionais, tendo incorporado as mais recentes inovações oriundas da pesquisa sobre estas linguagens, incluindo funções de alta ordem, semântica não-estrita, tipos algébricos definidos pelo usuário, suporte estático aos polimorfismos paramétrico e *ad hoc* (sobrecarga), casamento de padrões, compreensão de listas, sistema de módulos e um rico conjunto de tipos primitivos, incluindo arrays, inteiros de precisão fixa e arbitrária e números de ponto flutuante.

Nas subseções seguintes apresentamos diversos recursos relevantes disponibilizados por Haskell, que serão úteis para o entendimento dos programas apresentados em capítulos subseqüentes. Descrições mais abrangentes de Haskell podem ser encontradas em [BW88, Tho96, Hud00, Jon03].

3.1.1 Valores, tipos e declarações

Haskell é uma linguagem puramente funcional, o que implica que todas as computações são realizadas por meio da avaliação de expressões, com o intuito de produzir valores. Haskell é uma linguagem fortemente tipada, isto é, todos os valores possuem um tipo associado. Exemplos de expressões em Haskell incluem valores atômicos e funções, bem como valores estruturados como listas e tuplas. Todos os valores de Haskell são de "primeira-classe", podendo ser passados como argumentos para funções, retornados como resultado e utilizados em estruturas de dados.

A seguir temos exemplos de valores em Haskell, juntamente com os respectivos tipos:

```
5         :: Int         -- inteiro
'a'       :: Char        -- caractere
[1,2,3,4] :: [Int]       -- lista de inteiros
('a', 3)  :: (Char, Int) -- tupla de caractere e inteiro
```

Funções em Haskell são normalmente definidas por uma série de equações. Uma equação é um exemplo de declaração. Outra forma de declaração é a declaração de tipos, com a qual especifica-se explicitamente o tipo de uma expressão. As duas declarações a seguir definem a função *dec* que recebe um valor do tipo *Int* e retorna um valor deste mesmo tipo:

```
dec :: Int -> Int
dec n = n - 1
```

3.1.2 Tipos polimórficos

Haskell possui suporte a tipos polimórficos, que atuam em famílias inteiras de tipos. Por exemplo, $[a]$ é a família de tipos que abrange listas cujos elementos possuem tipo a. Listas de inteiros, de caracteres e mesmo listas de listas, são membros desta família. Os identificadores utilizados para representar tipos polimórficos são chamados de *variáveis de tipos* e devem ser iniciados com caracteres minúsculos para que possam ser distinguidos de tipos específicos.

Vejamos a seguir a função polimórfica *length* que retorna o número de elementos de uma lista de qualquer tipo:

```
length :: [a] -> Int
length [] = 0
length (x:xs) = 1 + length xs
```

A função é definida por meio de duas equações que utilizam *casamento de padrões*. No lado esquerdo das equações são especificados os padrões (neste caso [] e *x:xs*) que são casados contra o argumento em uma aplicação da função. Se o casamento de padrão contra um argumento tiver sucesso, o lado direito da equação será avaliado. Caso contrário, o processo de casamento de padrões continuará nas outras equações sucessivamente até que ocorra um casamento com sucesso. Neste exemplo a primeira equação é definida para listas vazias, enquanto que a segunda lida com listas não-vazias.

A função *length* é genérica (polimórfica), sendo desnecessário implementar versões idênticas para lidar com tipos específicos.

3.1.3 Tipos algébricos de dados

Tipos algébricos, úteis para representar valores definidos por enumerações ou pela composição de mais de um valor (*tipos produtos*), são especificados em Haskell por meio da declaração *data*, seguido do nome do tipo e de seus construtores.

A seguir temos um exemplo da utilização de tipos algébricos para modelar enumerações:

```
data Season = Spring | Summer | Autumn | Winter
```

O tipo *Season* possui quatro membros (*Spring*, *Summer*, *Autumn*, *Winter*) que são chamados de construtores de tipo.

A seguir temos um exemplo de um tipo produto:

```
data Person = Person String Int
```

que possui um único construtor. Para construir um valor deste tipo é necessário especificar o construtor *Person* seguido de um valor do tipo *String* e um do tipo *Int*:

```
Person "Carlos Andreazza" 25
```

3.1.4 Tipos sinônimos

Haskell provê, por conveniência, uma forma de declarar *tipos sinônimos*, que são nomes utilizados para referenciar tipos comumente usados. Tipos sinônimos são criados usando a declaração *type*. Vejamos alguns exemplos da utilização de tipos sinônimos:

```
type Name   = String
type Age    = Int
type Person = (Name,Age)
```

Tipos sinônimos não definem novos tipos; simplesmente ligam identificadores a tipos existentes. Por exemplo, o tipo $Person \rightarrow Name$ é equivalente a $(String, Int) \rightarrow String$.

3.1.5 Declaração newtype

Diferentemente de tipos sinônimos, o tipo criado pela declaração *newtype* possui uma identidade diferente do tipo original. A seguir temos um exemplo da utilização de *newtype*:

```
newtype Person = Person (String, Int)
```

Uma declaração *newtype* é similar a uma declaração *data*, diferindo pelo fato de apenas poder ser usada para definir tipos com um único construtor, o que lhe garante um maior desempenho, uma vez que não existe sobrecarga computacional na utilização de construtores.

3.1.6 Abstração lambda

Funções em Haskell não são definidas unicamente por meio de equações; uma forma alternativa é por meio de *abstrações lambda*. Uma abstração lambda é definida especificando-se os parâmetros formais da função após '\' (que é o caractere ASCII mais próximo da letra grega λ), então '->' e, finalmente, a definição da função. Como exemplo temos a seguir uma função que possui o mesmo comportamento da função *dec* definida anteriormente (Seção 3.1.1) por meio de equações:

```
\x -> x - 1
```

Abstrações lambda são utilizadas para criação de versões anônimas de funções, isto é, onde não é necessária a especificação de um identificador para referenciá-las.

3.1.7 Aplicação parcial

Aplicação parcial é uma técnica onde uma função é chamada com um número de parâmetros reais menor que o de parâmetros formais. O resultado de uma aplicação parcial é uma outra função que aguarda os argumentos restantes.

Como exemplo, consideremos a função *mult* a seguir que possui dois argumentos do tipo *Int*:

```
mult :: Int -> Int -> Int
mult x y = x * y
```

A expressão "*mult 2*" denota uma função que quando aplicada a um inteiro tem como resultado o dobro do valor fornecido:

```
(mult 2) 3 => 6
```

3.1.8 Funções de alta ordem

Funções de alta-ordem, como já mencionado, aceitam funções como argumento e/ou retornam uma função como resultado. A função *map*, que aplica uma função aos elementos de uma lista, é um exemplo de função de alta-ordem:

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

Outro exemplo de função de alta-ordem é uma aplicação parcial, uma vez que retorna uma função como resultado. Podemos, utilizando o exemplo da subseção anterior, multiplicar todos os elementos de uma lista por 2:

```
map (mult 2) [1,2,3] => [2,4,6]
```

3.1.9 Declarações locais

A definição de funções em Haskell pode incluir uma série de definições locais, isto é, visíveis apenas no escopo da função. Existem duas maneiras de fazer declarações locais em Haskell: empregando a expressão *where* ou a *let*. Declarações locais são interessantes quando sua utilidade se resume a apenas um local, evitando que o espaço de nomes fique sobrecarregado, provocando conflito entre identificadores declarados em um mesmo módulo.

Vejamos um exemplo:

```
sumSquares :: Int -> Int -> Int
sumSquares n m
  = sqN + sqM
   where
     sqN = n * n
     sqM = m * m
```

A função *sumSquares* retorna a soma dos quadrados de dois inteiros, utilizando as definições locais *sqN* e *sqM*.

Obtém-se resultado idêntico utilizando-se uma expressão *let*:

```
sumSquares :: Int -> Int -> Int
sumSquares n m
  = let sqN = n * n
        sqM = m * m
    in sqN + sqM
```

A diferença entre estas expressões é que *where* é apenas permitida no nível mais externo da definição de uma função, possuindo regras de visibilidade um pouco diferentes. Para maiores detalhes ver [Jon03].

3.1.10 Classes de tipos e sobrecarga

A diferença essencial entre funções sobrecarregadas e funções polimórficas é que, enquanto estas, por meio de uma única definição, operam uniformemente em todos os seus tipos, aquelas podem ser empregadas em uma variedade de tipos, mas com definições distintas para cada tipo específico.

Em Haskell, sobrecarga de funções (polimorfismo *ad hoc*) é realizada por meio de classes de tipos. Classes de tipos especificam assinaturas de funções que caracterizam uma classe. Para incluir um tipo em particular como membro de uma classe é necessária a criação de uma instância desta classe, juntamente com as implementações das funções associadas a ela.

Para ilustrar, vejamos como é possível definir o tipo *Person* (Seção 3.1.5) como instância da classe *Eq* que define relações de igualdade entre valores de um tipo:

```
class Eq a where
 (==) :: a -> a -> Bool
 (/=) :: a -> a -> Bool

instance Eq Person where
  Person (nm, age) == Person (nm', age')
        = (nm == nm') && (age == age')
  p1 /= p2 = not (p1 == p2)
```

A definição de uma classe pode incluir implementações padrão para as funções da classe. Se uma função de uma classe é omitida na declaração de instância, então, caso exista uma implementação padrão, esta é utilizada.

Haskell disponibiliza bibliotecas com diversas classes predefinidas (*Eq* é uma delas), além de ser permitida a especificação de novas classes e instâncias pelo usuário.

3.1.11 Classes de construtores

É possível generalizar o conceito de classes de tipos para que passem a agrupar não apenas tipos, mas também construtores de tipos (funções que constroem tipos). Esta generalização é conhecida como *classes de construtores* [Jon95] [1].

A declaração a seguir define a classe de construtores *Functor*:

[1] Publicações mais recentes, entretanto, não fazem distinção entres estes conceitos, empregando para ambos tão-somente o termo "classes de tipos".

```
class Functor f where
  map :: (a -> b) -> (f a -> f b)
```

O parâmetro f da classe *Functor* é um construtor de tipo que, no caso, é aplicado aos tipos a e b.

Classes de construtores desempenham papel importante na definição de mônadas e transformadores monádicos, apresentados no próximo capítulo.

3.1.12 Módulos

Um programa em Haskell é constituído por um conjunto de módulos. Cada módulo possui uma série de declarações: de tipos, funções, classes, etc. É possível manter algumas destas definições invisíveis fora do módulo, sendo empregadas apenas internamente. As definições exportadas pelo módulo e, portanto, visíveis externamente, constituem sua interface. Além disso, um módulo pode utilizar as definições exportadas por outros módulos através de declarações de *import*.

Um exemplo de módulo Haskell é visto a seguir:

```
module Bee (beeKeeper, module Ant)
import Ant
(...)
```

Aqui temos a declaração do módulo *Bee* que importa as definições do módulo *Ant*. A interface do módulo é constituída da definição *beeKeeper* e das declarações visíveis do módulo *Ant*. Todas as outras definições são mantidas invisíveis fora do módulo.

CAPÍTULO 4

MÔNADAS

Linguagens puramente funcionais oferecem diversas vantagens no que diz respeito a um desenvolvimento rigoroso de software, uma vez que a forma como são projetadas torna viável um entendimento matemático dos programas desenvolvidos, facilitando a prova de propriedades nos mesmos.

Contudo, tais linguagens não interagem de forma adequada com as chamadas *funcionalidades impuras*. Uma funcionalidade é considerada impura quando não pode ser vista como abreviação para alguma construção do λ-cálculo (e, portanto, não pode ser analisada usando a teoria do λ-cálculo) [Gor88]. Exemplos de funcionalidades impuras incluem interações de I/O, controle de exceções, estado entre computações e não-determinismo.

Para superar os obstáculos que a implementação de funcionalidades impuras em linguagens funcionais impõem aos programadores, pesquisadores empregaram o conceito de mônada. Mônadas permitem que uma funcionalidade impura seja realizada em programas funcionais sem abdicar das vantagens oferecidas pela transparência referencial.

Este capítulo apresenta o conceito de mônada dentro do contexto de linguagens funcionais. Apresentamos os passos para definição de uma mônada em Haskell, juntamente com descrições das principais mônadas utilizadas na prática (Seção 4.1) e a conveniência sintática disponibilizada por Haskell para a programação monádica (Seção 4.2). Expomos também as principais dificuldades encontradas para combinar duas mônadas quando se deseja realizar simultaneamente as funcionalidades providas por ambas (Seção 4.3). Por fim, na Seção 4.4, analisamos os transformadores monádicos, que representam uma alternativa razoável às dificuldades oferecidas pelo processo de combinação manual de mônadas.

4.1 CONCEITO

O conceito de mônada – uma idéia que tem sua origem na teoria das categorias – foi utilizado por Moggi [Mog89] para estruturar a semântica denotacional de linguagens de programação. Wadler [Wad90, Wad92a, Wad92b] foi o pioneiro na utilização de mônadas para estruturar programas funcionais.

Desde então, mônadas tornaram-se ferramentas práticas importantes em linguagens funcionais. A razão disso é que mônadas provêem um *framework* uniforme para descrever funcionalidades impuras, sem a necessidade de renunciar às propriedades de uma linguagem puramente funcional.

Mônada é uma forma de estruturar computações em termos de valores e seqüências de computações que fazem uso destes valores. Mônadas permitem ao programador desenvolver computações utilizando blocos de construções seqüenciais. A mônada determina como a combinação destas computações constitui uma nova computação, liberando

o programador de codificar a combinação manualmente cada vez que a mesma é solicitada. Desta forma, uma mônada pode ser entendida como uma estratégia para combinar computações com o objetivo de definir computações mais complexas.

Mônadas são ferramentas úteis para estruturar programas funcionais. Elas possuem três propriedades que as tornam especialmente vantajosas [New04]:

i) **Modularização**. Mônadas possibilitam a composição de computações a partir de computações mais simples; permitem também a separação entre a estratégia de combinação e a execução propriamente dita das computações.

ii) **Flexibilidade**. Programas funcionais estruturados por meio de mônadas são mais fáceis de adaptar que os que não as utilizam. Isto porque a estratégia computacional especificada na mônada está definida em um único local, ao invés de ficar distribuída por todo o programa.

iii) **Isolamento**. Mônadas podem ser utilizadas para criar estruturas computacionais com estilo imperativo que, entretanto, permanecem isoladas do corpo principal do programa funcional. Isto é útil para incorporação de efeitos colaterais (tais como I/O) e modelar estado (que destrói a transparência referencial) em linguagens puramente funcionais como Haskell.

4.1.1 Mônadas em Haskell

Uma mônada é definida como uma família de tipos $m\ a$, baseada no construtor de tipos polimórfico m, e duas funções associadas, *return* e >>=. A função *return* não realiza computações, apenas constrói valores do tipo m. Já a função >>= combina duas computações sucessivas, definindo uma computação mais complexa. Em outras palavras, >>= representa a estratégia computacional realizada pela mônada.

A classe predefinida *Monad* de Haskell (mais precisamente, uma classe de construtores – ver Seção 3.1.11) é utilizada para especificação de mônadas. Sua definição, apresentada a seguir, possui duas outras funções, além de *return* e >>=:

```
class Monad m where
  return :: a -> m
  (>>=)  :: m a -> (a -> m b) -> m b
  (>>)   :: m a -> m b -> m b
  fail   :: String -> m a
```

A função >> é utilizada para combinar computações que não transmitem valores entre si, enquanto que *fail* corresponde a uma falha na execução de uma computação. A definição de *Monad* contém implementações padrão para as funções >> e *fail*:

```
m >> k = m >>= (\_ -> k)
fail s = error s
```

Observe que a definição de >> é feita em termos de >>=, onde o valor retornado pela primeira computação é descartado. *fail* exibe uma mensagem de erro e encerra o programa.

Qualquer construtor de tipo m pode ser considerado uma mônada, uma vez que se defina o mesmo como instância da classe *Monad*. Além disso, as operações *return* e >>= devem obedecer às seguintes leis (conhecidas como *axiomas monádicos*):

```
m         >>= return = m
return x  >>= f      = f x
(m >>= f) >>= g      = m >>= (\x -> f x >>= g)
```

As duas primeiras leis estabelecem a função *return* como identidade relativamente à função >>=, enquanto que a última é uma lei de associatividade de >>=. É de responsabilidade do programador definir as operações *return* e >>= de tal forma que os axiomas monádicos se apliquem.

Nas próximas subseções apresentamos exemplos de mônadas freqüentemente utilizadas na prática que são disponibilizadas por implementações de Haskell.

4.1.2 Mônada Identity

A mônada *Identity* não incorpora qualquer estratégia de computação. Ela simplesmente aplica a função definida pela computação subseqüente ao valor retornado pela computação anterior sem qualquer modificação.

```
newtype Identity a = Identity a

instance Monad Identity where
    return a           = Identity a
    (Identity x) >>= f = f x
```

Computacionalmente esta mônada representa um estado em que nenhuma ação é desempenhada e os valores são retornados imediatamente.

4.1.3 Mônada []

A definição da mônada [] (lista) é vista a seguir:

```
instance Monad [] where
  return x = [x]
  xs >>= f = concat (map f xs)
  fail s   = []
```

A interpretação de uma mônada lista pode ser a de uma computação não-determinística: os elementos de uma lista representam todos os resultados possíveis de uma computação não-determinística. Neste caso *return* oferece uma única resposta, enquanto que >>= aplica a função f (computação subseqüente) a todos os elementos (possíveis resultados) de xs e concatena os resultados possíveis em uma única lista. Em caso de falha, a computação não-determinística retorna nenhum valor (*fail s = []*).

4.1.4 Mônada Maybe

Consideremos a seguir o tipo de dado *Maybe* de Haskell:

```
data Maybe a = Just a | Nothing
```

que representa computações que podem falhar ao retornar um resultado. O tipo *Maybe* sugere uma estratégia para combinar computações que retornam valores deste tipo: se uma computação composta consiste de duas subcomputações onde a segunda depende do valor retornado pela primeira, então a combinação deve retornar *Nothing* quando pelo menos uma das subcomputações retornar *Nothing*; quando as subcomputações retornam um valor normalmente, a combinação delas deve retornar o resultado da aplicação da segunda computação ao valor retornado pela primeira.

A seguir temos a declaração que define o tipo de dado *Maybe* como instância de *Monad*:

```
instance Monad Maybe where
  Just x   >>= f = f x
  Nothing  >>= f = Nothing
  return x       = Just x
```

A operação $>>=$ funciona da seguinte maneira: se a computação anterior produziu um valor (*Just x*), este é passado para a computação seguinte *f*. Caso contrário, o estado de erro (*Nothing*) será propagado. A função *return* injeta um valor na mônada *Maybe*.

Consideremos a utilização da mônada *Maybe* para capturar exceções em uma função que avalia expressões aritméticas. A seguir apresentamos o tipo de dado que representa expressões aritméticas e a função de avaliação correspondente, extraídos de [Wad92a].

```
data Exp = Con Int | Plus Exp Exp | Div Exp Exp

eval :: Exp -> Maybe Int

eval (Con x) = return x

eval (Plus x y) = eval x >>= \i ->
                  eval y >>= \j ->
                  return (i+j)

eval (Div x y)  = eval x >>= \i ->
                  eval y >>= \j ->
                  if j==0 then Nothing
                          else return (i `div` j)
```

Os valores calculados recursivamente por *eval* têm de ser ligados às variáveis i e j (processo inverso ao da função *return*) que podem ser utilizadas em operações com inteiros. As duas últimas linhas são responsáveis por capturar exceções, evitando que, por exemplo, a avaliação da expressão *Div (Con 1) (Con 0)* seja interrompida com um erro de execução.

4.1.5 Mônada IO

As interações entre programas Haskell e o mundo exterior são realizadas por meio da mônada *IO*. Uma expressão *IO a* é um programa que realiza alguma operação de I/O, após o que retorna um valor do tipo *a*. A declaração que define *IO* como instância de *Monad* é definida internamente, já que depende do estado do sistema operacional.

Haskell provê funções primitivas que realizam operações de I/O: o seqüenciamento destas operações é feito por meio das funções >> e >>=. As funções *getLine* e *putStr* são exemplos de operações primitivas:

```
getLine :: IO String
putStr :: String -> IO ()
```

A função *getLine* lê uma linha de texto da entrada padrão e retorna um valor *String*; *putStr* recebe um valor *String* como argumento e o escreve na saída padrão.

O programa a seguir exemplifica o seqüenciamento entre duas operações de *IO*:

```
main = getLine >>= \s ->
       putStr s
```

onde uma *String* é lida por *getLine* e seu resultado é ligado ao identificador *s* e então passado para *putStr* que escreve seu valor na saída padrão.

A função *getChar* lê um caractere da entrada padrão e retorna um valor do tipo *Char*. Utilizemos *getChar* para definir uma função que ler um número dado de caracteres na entrada padrão e os retorne como um valor do tipo *String*:

```
gets :: Int -> IO String
gets 0     = return []
gets (i+1) = getChar >>= \c ->
             gets i  >>= \s ->
             return (c:s)
```

A função *gets* sequencia a função *getChar* com suas chamadas recursivas, retornando ao final a *String* lida.

É permitida a definição de combinadores para operações de *I/O*. Como exemplo temos a seguir uma função que possui como argumento uma lista de "comandos" (operações de *I/O*) que produz uma única operação de *I/O* e retorna uma lista com os valores calculados pelas operações da lista de entrada:

```
prod :: [IO a] -> IO [a]
prod []     = return []
prod (m:ms) = m >>= \x ->
              prod ms >>= \xs ->
              return (x:xs)
```

Por meio de *prod* redefinimos a seguir a função *gets* utilizando funções de alta-ordem:

```
gets i = prod (take i (repeat getc))
```

A função *take i xs* obtém os primeiros *i* elementos da lista *xs* e *repeat x* retorna uma lista infinita com a operação *x*. Assim, a definição de *gets* é feita obtendo-se os *i* primeiros elementos de uma lista infinita de seqüências de comandos *getc*.

4.2 NOTAÇÃO DO

Haskell provê a notação *do* como uma conveniência sintática para programação monádica. As computações monádicas que utilizam *do* são mais legíveis uma vez que eliminam a necessidade de chamadas às funções >> e >>= e a utilização de abstrações lambda para passagem de valores entre computações consecutivas. Programas que fazem uso desta notação possuem um estilo, por assim dizer, imperativo, adequado à situação. Qualquer instância da classe *Monad* pode fazer uso de expressões *do*.

Vejamos, por exemplo, como fica o avaliador de expressões visto na seção anterior utilizando a notação *do*:

```
eval :: Exp -> Maybe Int

eval (Con x)    = return x

eval (Plus x y) = do i <- eval x
                     j <- eval y
                     return (i+j)

eval (Div x y)  = do i <- eval x
                     j <- eval y
                     if j==0 then Nothing
                             else return (i `div` j)
```

Expressões que utilizam *do* são traduzidas em chamadas à função >>= da seguinte forma [Jon03]:

```
do { e }              = e
do { e; stmts }       = e >>= \_ -> do { stmts }
do { p <- e; stmts }  = let ok p = do {stmts}
                            ok _ = fail "..."
                        in e >>= ok
do {let decls; stmts} = let decls
                        in do {stmts}
```

As reticências "..." simbolizam a mensagem de erro, que é passada para a função *fail*.

É importante ressaltar que para um correto funcionamento da notação *do* a instância particular da classe *Monad* deve respeitar os axiomas monádicos.

4.3 COMBINAÇÃO DE MÔNADAS

É freqüente, na prática, a necessidade da combinação de duas mônadas para suportar simultaneamente as funcionalidades providas por elas. A construção de uma mônada a partir da combinação de outras duas, entretanto, normalmente não é uma tarefa trivial [KW92]. Desta forma, seria útil ter-se à mão uma forma sistemática de combinar mônadas. Este método, no entanto, não existe; contudo, é concebível que se tenha uma biblioteca de mônadas primitivas e, para cada mônada, uma técnica para combiná-la com outras; tal biblioteca pode, eventualmente, ser dividida em coleções de mônadas que possam ser combinadas utilizando-se a mesma estratégia.

Analisemos agora um exemplo que ilustra as dificuldades encontradas durante o processo de combinação entre mônadas. Utilizaremos duas mônadas, uma das quais é *Maybe* (discutida anteriormente na Seção 4.1.4); a outra é a mônada *St*. O tipo de dado *St* é visto a seguir:

```
newtype St a = St (State -> (a, State))
```

O tipo *St*, que representa uma mônada de estado em um tipo *a*, é uma função que recebe um estado e retorna um par, constituído de um valor do tipo *a* e um novo estado. Este modelo de estado pode ser utilizado, por exemplo, no avaliador de expressões visto na Seção 4.1.4 para computar o número de operações de divisão desempenhadas.

Existem duas maneiras distintas de combinar as mônadas *St* e *Maybe*. Vejamos a seguir as alternativas:

```
newtype StMaybe a = St (State -> (Maybe a, State))
```

ou

```
newtype MaybeSt a = St (State -> Maybe (a, State))
```

cada uma das quais possuindo significados diferentes. No primeiro modelo, a avaliação de uma expressão que levanta uma exceção também retorna um estado, o que pode ser útil em situações onde é importante a manutenção do estado após exceções. No segundo modelo, o estado não é mantido quando um erro ocorre.

Este método de combinação de mônadas mostra-se bastante insatisfatório na prática, uma vez que a combinação de determinadas mônadas pode se tornar muito complexa ou mesmo impraticável. Na próxima seção veremos uma maneira sistemática de realizar novas funcionalidades em uma mônada arbitrária.

4.4 TRANSFORMADORES MONÁDICOS

Esta seção explora o conceito de transformadores monádicos, que permitem adicionar operações (isto é, introduzir novas funcionalidades) a uma mônada.

Um transformador monádico adiciona novas funcionalidades a uma mônada sem modificar a natureza da mesma. Transformadores monádicos, como apresentados em [LHJ95], são implementados por meio de classes de construtores.

Um transformador monádico pode ser definido como qualquer construtor de tipo t tal que se m é uma mônada, então "$t\ m$" também o é. É possível expressar esta definição por meio da classe de construtores *MonadT* a seguir [1]:

```
class (Monad m, Monad (t m)) => MonadT t m where
  lift :: m a -> t m a
```

O termo *(Monad m, Monad (t m))*, que aparece à direita do operador =>, é utilizado para especificar que os tipos m e t são instâncias da classe *Monad*. A função *lift* embute a computação definida na mônada m dentro da mônada "$t\ m$".

Para ilustrar o conceito de transformadores monádicos vejamos como é possível adicionar a noção de estado para uma mônada qualquer. A seguir temos a definição do tipo *StateT*:

```
newtype StateT s m a = StateT (s -> m (s, a))
```

onde m é um construtor de tipo (fato automaticamente determinado pelo sistema de tipos).

Agora, por meio de uma declaração de instância de classe, desejamos declarar que *StateT s m* é uma mônada (dado que m também o é), e que *StateT s* é um transformador monádico.

A seguir vemos a definição monádica para *StateT s m*, que implementa as funções *return* e >>=:

```
instance Monad m => Monad (StateT s m) where
  return x = \s -> return (s, x)
  m >>= k  = \s0 -> m s0 >>= \(s1, a) -> k a s1
```

Deve-se ressaltar que estas definições não são recursivas; o sistema de classes de construtores infere automaticamente que as funções *return* e >>= que aparecem no lado direito das definições são da mônada m.

A definição de *StateT s* como transformador monádico é vista a seguir:

```
instance (Monad m, Monad (StateT s m)) =>
                            MonadT (StateT s) m where
  lift m = \s -> m >>= \x -> return (s, x)
```

A função *lift* executa a computação m em um novo contexto, preservando o estado.

Por fim, uma mônada de estado deve suportar uma operação para atualizar seu estado. Desta forma, para manter a implementação modular, definimos uma nova classe *StateMonad*, que possui a função *update*:

```
class Monad m => StateMonad s m where
  update :: (s -> s) -> m s
```

[1] Os exemplos desta seção, apresentados em [LHJ95], utilizam Gofer [Jon94] que possui sintaxe similar a Haskell. Na época da publicação alguns dos recursos presentes em Gofer, como classes de construtores, ainda não estavam implementados em Haskell.

A declaração de instância a seguir define que *StateMonad* transforma qualquer mônada em uma mônada de estado, onde a função *update* aplica a função *f* ao estado, retornando o estado anterior:

```
instance Monad m => StateMonad s (StateT s m) where
  update f = \s -> return (f s, s)
```

Adicionar novas funcionalidades em uma mônada por meio de transformadores monádicos resulta em código mais claro e simples, uma vez que não há necessidade de manipular explicitamente os tipos da mônada: a função *lift* cuida destes detalhes.

CAPÍTULO 5

ASPECTH

O emprego de mônadas possui papel fundamental na estruturação de programas Haskell. Basta lembrar de que, para um programa realizar qualquer interação de *I/O*, o mesmo deve fazer uso de uma mônada (*IO*). Ademais, necessidades referentes a funcionalidades impuras, tais como manipulação de estado e exceções, são freqüentes durante a implementação de soluções.

Observa-se na prática que programas que utilizam construções monádicas, por vezes, possuem os mesmos inconvenientes provocados pela modularização inadequada ou insuficiente das funcionalidades implementadas por elas (e que foram discutidas no Capítulo 2): legibilidade comprometida; mapeamento inadequado entre uma funcionalidade específica e o código que a implementa; dificuldade de reuso de código e; evolução custosa. Em outras palavras, os mesmos tipos de *crosscutting concerns* encontrados, por exemplo, em programas orientados a objetos, estão presentes em programas Haskell que utilizam mônadas.

É interessante, portanto, investigar até que ponto os conceitos e abstrações relacionados com Programação Orientada a Aspectos, que já demonstraram seu valor no tocante a melhorias na separação de preocupações, por exemplo, em programas orientados a objetos, podem também ser úteis dentro do contexto de programação funcional e, em particular, de programas estruturados por meio de mônadas.

AspectH, uma extensão orientada a aspectos da linguagem Haskell, representa um passo importante no sentido de demonstrar que AOP pode fazer por Haskell e pela programação funcional o que já faz por linguagens orientadas a objetos, oferecendo ao programador novas construções que o ajudem a escrever um código de melhor qualidade, mais fácil de ler e modificar, bem como mais propício para o reuso em outras aplicações.

A similaridade entre as linguagens AspectH e AspectJ (Seção 2.3.1) não se restringe simplesmente a seus nomes: AspectH implementa o mesmo mecanismo AOP encontrado em AspectJ, possuindo abstrações e nomenclatura semelhantes.

O objetivo deste capítulo é apresentar a linguagem AspectH. Inicialmente damos uma visão geral da linguagem (Seção 5.1). Discutimos seu modelo de *join points*, que esclarece os pontos de um programa que são passíveis a execução de código advindo de aspectos (Seção 5.2). Continuando, temos uma descrição dos *pointcuts* da linguagem, com explicações sobre as características dos operadores primitivos e de como é feita a composição deles para definir padrões de *join points* mais elaborados (Seção 5.3). As declarações de *advice* vem a seguir, onde temos suas características principais, os tipos básicos e a espécie de composição que elas determinam, bem como uma exposição de como é feito o uso de informações de contexto presentes nos *join points* selecionados (Seção 5.4). Então, neste ponto, já possuímos as informações necessárias para examinar a definição das unidades modulares de AspectH, os aspectos, onde são implementados os *crosscutting concerns* de um sistema (Seção 5.5), o importante tópico referente a

ordem relativa de execução do código dos aspectos em um mesmo *join point* (Seção 5.6) e uma descrição global do processo de combinação (Seção 5.7). Por fim, na Seção 5.8, exibimos um pequeno exemplo que explora a implementação de *crosscutting concerns* de programas monádicos em aspectos, por meio de duas versões que atuam em tipos distintos de *join points*.

5.1 VISÃO GERAL DE ASPECTH

AspectH estende a linguagem funcional Haskell com construções AOP, propiciando novos meios de separar preocupações em programas monádicos. O mecanismo AOP implementado pela linguagem é o de *pointcuts* e *advice* como em AspectJ: através da definição de *pointcuts* é possível interceptar os *join points* de um programa e especificar código adicional (*advice*) que será executado lá.

As unidades modulares de AspectH são os aspectos que capturam a implementação de *crosscutting concerns* em programas Haskell estruturados por meio de mônadas. A definição de um aspecto em AspectH é feita de forma semelhante a um módulo Haskell (Seção 3.1.12), onde, além de todas as declarações de Haskell, temos a possibilidade de definir declarações de *advice*.

Uma declaração de *advice* em AspectH especifica uma computação que executará em determinados *join points* de um programa. Ela é composta de três partes, a primeira das quais define seu tipo (isto é, a maneira como é feita a combinação com os *join points*), a segunda é um *pointcut* que especifica os *join points* afetados, enquanto que a terceira representa a computação propriamente dita (uma expressão padrão de Haskell).

Um *pointcut* "escolhe" de forma declarativa os pontos de interesse em um programa; é uma linguagem de operadores primitivos e de composição que expressa em alto nível um conjunto de *join points* envolvidos diretamente com a implementação de um *crosscutting concern*. Em outras palavras, *pointcuts* representam predicados acerca de pontos na execução de um programa: todo código que atender às condições requeridas pelo predicado será selecionado pelo *pointcut*. Os *pointcuts* de um aspecto, portanto, definem as regras de combinação (*weaving rules*) que são utilizadas para combinar os aspectos com o programa base.

Pointcuts e *advice* são entidades intrinsecamente ligadas em AspectH, uma vez que cada *advice* é especificado juntamente com o *pointcut* que indica o código cuja semântica é influenciada pelo código definido no "lado direito" do *advice*. Além disso, as variáveis (metavariáveis, mais exatamente) definidas no *pointcut* podem ser utilizadas dentro do escopo da computação definida pelo *advice*. Assim, as informações de contexto contidas nos *join points* interceptados não são perdidas.

É freqüente na definição de aspectos a situação em que mais de um aspecto intercepte um mesmo *join point*, ou até mesmo mais de uma declaração de *advice* em um mesmo aspecto. Neste caso é necessário que exista uma ordenação uniforme que será usada pelo *Weaver*. AspectH utiliza uma estratégia arbitrária, que será discutida mais a frente, neste capítulo.

AspectH atua basicamente em código estruturado por meio de mônadas. Esta é uma decisão de projeto importante: não há como modificar código não-monádico. A modificação de código determinada por uma declaração de *advice*, onde uma ordem

de execução é especificada com relação ao *join point*, não faz sentido em uma função que tão-somente retorna uma expressão, principalmente considerando que não existem efeitos colaterais em expressões Haskell. Assim, a abrangência do código dos aspectos em AspectH é restrita especificamente a funções cujo tipo de retorno é instância de *Monad*. A natureza seqüencial pertinente ao conceito de mônada o torna um ambiente adequado para abstrações tais como declarações de *advice*, que especificam explicitamente a ordem de execução do código definido por elas com relação aos *join points* que elas interceptam.

Pelo fato de modificar código de funções monádicas, a linguagem se presta a definição de *ações monádicas*: uma ação monádica é qualquer código que se deseja adicionar em um código monádico, com o intuito de realizar determinada operação. Em AspectH, a computação de uma declaração de *advice* corresponde a uma ação monádica.

Em AspectH os aspectos contêm as regras de transformação (*pointcuts*) que serão realizadas pelo *Weaver*. Esta interpretação é importante para definição de um método de implementação onde o código base contenha apenas a "funcionalidade básica" do sistema e os aspectos transformem incrementalmente este código para definição do sistema final, que envolve os *crosscutting concerns* especificados separadamente em aspectos. Além disso, os aspectos podem ser removidos do sistema automaticamente, bastando apenas que o *Weaver* seja executado sem a presença do aspecto que se deseja remover.

É interessante analisar a aderência de AspectH a AOP no que diz respeito às propriedades clássicas desta técnica de programação. Quanto a quantificação, tem-se que as declarações de *advice* constituem o meio de especificar de forma declarativa diversos pontos em um programa que serão influenciados por seu código. No que se refere a *obliviousness*, a inconsciência pregada por esta propriedade no contexto de AspectH, considerando principalmente a decisão de projeto que restringe sua atuação a funções monádicas, torna-se de certo modo impraticável. É impossível escrever os aspectos sem ter em conta a espécie de código que se pretende modificar, ou seja, se tal código é monádico ou estritamente funcional. Assim, ao implementar um sistema em AspectH é necessário que o programador tenha sim consciência de relacionamentos entre o programa base e os aspectos para constatar a coerência do mesmo quanto as restrições impostas pela linguagem.

5.2 MODELO DE JOIN POINTS

Como mencionado na Seção 2.1, o modelo de *join points* provê uma referência comum que possibilita a execução dos aspectos e do programa base de forma coordenada. O modelo de *join points* determina os pontos de um programa que estão sujeitos à influência do código definido nos aspectos. *Join points* em AspectH são pontos bem definidos na execução de um programa.

Existem dois tipos de *join points* que podem ser identificados em AspectH:

- Equações que definem funções, isto é, a execução de funções e
- chamadas de funções.

A identificação destes *join points* é feita indistintamente em módulos convencionais e aspectos, isto é, aspectos podem modificar código tanto de módulos Haskell quanto de outros aspectos.

É interessante ressaltar que estes dois tipos de *join points* permitidos dizem respeito apenas a código monádico, isto é, tanto no caso de execução, quanto de chamadas, devem ser referentes a funções monádicas (cujo tipo de retorno seja uma instância da classe *Monad*).

Outro ponto importante a ser observado é que os *join points* podem ser interceptados tanto em código que utiliza chamadas explícitas às funções >> e >>=, quanto o que faz uso da notação *do*.

5.2.1 Execução de funções

No Capítulo 3 vimos que funções em Haskell podem ser definidas por várias equações; a equação adequada será determinada por meio de casamento de padrões. AspectH disponibiliza meios para identificar individualmente uma equação específica ou todas, de maneira simultânea. A identificação é feita com base no nome da função e nos padrões que definem os parâmetros formais da equação. Entretanto, é possível interceptar a execução de funções definidas anonimamente por meio de abstrações lambda.

Uma condição referente à seleção de um *join point* é que o mesmo deve ser referente a uma função exportada pelo módulo onde foi definida, isto é, que a função seja visível fora do módulo. O processo de combinação será interrompido caso uma declaração de *advice* selecione equações de uma função visível apenas dentro do módulo que a define. Esta decisão foi tomada com o intuito de preservar as propriedades de *information hiding* de Haskell.

Consideremos a função *eval* abaixo (vista na Seção 4.2), que possui um parâmetro formal, tipo de retorno *Maybe Int* (instância de *Monad*) e é definida por três equações (correspondentes a cada um dos construtores do tipo de dado *Exp*):

```
data Exp = Con Int | Plus Exp Exp | Div Exp Exp

eval :: Exp -> Maybe Int

eval (Con x)    = return x

eval (Plus x y) = do i <- eval x
                     j <- eval y
                     return (i + j)

eval (Div x y)  = do i <- eval x
                     j <- eval y
                     if j==0 then Nothing
                             else return (i `div` j)
```

A execução de cada uma destas três equações representa um *join point* válido, existindo meios em AspectH de identificá-las de maneira simultânea (baseada no identificador *eval*) ou individualmente (utilizando-se cada um dos construtores de *Exp*).

5.2.2 Chamadas de funções

O outro tipo de *join point* permitido em AspectH são chamadas de funções. AspectH permite a seleção de chamadas a funções cujo tipo de retorno seja instância da classe *Monad*. As chamadas de função são selecionadas através de seus identificadores, bem como através de casamento de padrões contra os parâmetros reais da chamada.

Por exemplo, na função *eval* apresentada acima, as chamadas de funções que podem ser identificadas são as feitas à função *return* e as chamadas recursivas feitas a função *eval*.

Como veremos na Seção 5.8, a especificação de computação adicional referente a uma função produz um resultado externo independente do tipo de *join point* escolhido – se nas equações ou nas chamadas da função. No entanto, selecionar as equações garante que tal computação será executada sempre que ocorrer uma chamada para esta função; interceptar uma chamada restringe o efeito desejado apenas a este ponto.

5.3 POINTCUTS

Um *pointcut* define um conjunto de *join points* que contém, opcionalmente, valores no contexto de execução destes *join points*. Em AspectH, a definição de um *pointcut* é feita anonimamente, juntamente com a declaração de *advice* correspondente. Como em AspectJ, *pointcuts* em AspectH não são de alta ordem nem paramétricos.

Os *pointcuts* selecionam os *join points* desejados em um programa por meio de um conjunto de operadores primitivos que podem ser compostos para definir padrões de *join points* mais elaborados.

Pointcuts representam o meio de especificar declarações quantificadas em AspectH, isto é, de definir as condições que, uma vez satisfeitas durante a execução do programa base, devem resultar na execução de código localizado em aspectos. Esta propriedade, quantificação (Seção 2.2.1), é uma das propriedades clássicas das instâncias de AOP.

Um *pointcut* seleciona a execução de chamadas de funções através da comparação de identificadores, casamento do número e dos padrões dos argumentos e pela inspeção de definições aninhadas. Este último caso é referente à identificação de *join points* localizados dentro de uma equação de função. Assim, é possível restringir as chamadas a uma função que serão selecionadas com base na equação onde é feita a chamada; da mesma maneira é possível selecionar a execução de uma função definida localmente em outra função. Em ambos os casos, não existe uma limitação quanto ao nível de aninhamento de um *join point* escolhido em uma equação.

5.3.1 Operadores primitivos

Os operadores primitivos de *pointcut* são responsáveis pela seleção dos dois tipos de *join points* permitidos em situações específicas, correspondentes às diversas maneiras em que é feita a identificação. AspectH possui quatro operadores primitivos de *pointcut*, a saber, *call*, *execution*, *args* e *within*.

O operador *call* seleciona as chamadas de uma função dentro de um módulo, com base no identificador da função. Vejamos a seguir um exemplo de seu uso:

```
call Module.functionName
```

Este *pointcut* seleciona todas as chamadas à função *functionName* localizadas no módulo *Module*. É interessante ressaltar este fato: o nome do módulo especificado diz respeito ao local onde a chamada é feita e não onde a função foi definida. Assim, caso seja necessário selecionar todas as chamadas feitas a uma função, é preciso que todos os módulos onde estas chamadas ocorram sejam identificados separadamente em um *call* e, então, combinados [1]. Mais a frente veremos como é feita a combinação de operadores primitivos.

Não é estritamente necessário especificar o identificador completo da função cuja chamada se deseja selecionar. Pode-se indicar apenas seus caracteres iniciais. Isto pode ser útil quando a implementação segue um padrão de nomeação, permitindo selecionar chamadas a um conjunto de funções cujos identificadores possuem um prefixo em comum, por meio de apenas um *call*. A seguir temos um exemplo desta possibilidade:

```
call Module.function **
```

A presença do "coringa" ** indica que todas as chamadas a funções cujos identificadores iniciem com a seqüência *function* são selecionadas. Os operadores *execution* e *within*, que especificam nomes de funções do mesmo modo que *call*, também permitem o uso do coringa.

O operador *execution* seleciona a execução de uma função, da mesma maneira que *call*, através de seu identificador. A seguir temos um exemplo da utilização deste operador:

```
execution Module.functionName
```

Os *join points* selecionados por este *pointcut* são todas as equações que definem a função *functionName* definida no módulo *Module*. Como destacado na Seção 5.2, as funções interceptadas por este operador devem ser exportadas pelo módulo.

O operador *args* seleciona a execução e chamadas de funções cujos parâmetros formais (no caso de equações) ou reais (no caso de chamadas) casem com o número e os padrões especificados pelo operador. A informação de contexto dos *join points* selecionados é ligada aos identificadores que definem os padrões. Assim, este operador (bem como o operador *within*, visto a seguir) expõe informações que podem ser utilizadas dentro do escopo da computação definida pelo *advice*. Como exemplo, o *pointcut* seguinte seleciona todos os *join points* que possuam dois argumentos:

```
args id1 id2
```

[1] Esta provavelmente não é a semântica esperada para este operador e, de certo, representa uma restrição que limita a expressividade nos *pointcuts*. A decisão foi tomada por razões técnicas, uma vez que o pré-processador que implementa a linguagem não possui informações suficientes quanto ao módulo onde um identificador foi declarado.

Ademais, os identificadores *id1* e *id2* serão ligados aos parâmetros (formais ou reais) dos *join points* selecionados.

As regras para casamento de padrões no operador *args* são as mesmas definidas normalmente em programas Haskell [2]. Portanto, é possível especificar padrões que casam com literais, listas, construtores de tipo, bem como o padrão coringa _ que casa com qualquer padrão, não resultando, contudo, em ligações. A seguir apresentamos um exemplo de *pointcut* que seleciona equações e chamadas de funções que possuam dois padrões, onde o primeiro é uma lista com apenas um elemento (que será ligado ao identificador *a*), podendo o segundo ser qualquer expressão (devido a utilização do padrão coringa):

```
args [a] _
```

Por fim, o operador *within* seleciona todos os *join points* – tanto chamadas quanto execução de funções – dentro de uma equação que define uma função. Este operador especifica, além do nome da função os padrões definidos por suas equações. Desta forma, para que uma equação seja identificada, a mesma deve casar com o nome e com os padrões especificados pelo operador. Do mesmo modo que o operador *args*, *within* possibilita a exposição de informações de contexto: quando ocorre um casamento com sucesso entre os padrões da equação e os especificados no *within*, os identificadores definidos neste operador são ligados de forma adequada aos valores correspondentes. Vejamos um exemplo de uso do operador *within*:

```
within Module.functionName id1 id2
```

Este *pointcut* seleciona todos os *join points* dentro de equações da função *functionName* (que deve ter dois argumentos) do módulo *Module*. Os identificadores *id1* e *id2* serão ligados aos parâmetros formais da função *functionName*; observe que esta informação de contexto é referente à função *functionName* e não aos *join points* interceptados por *within*.

A Tabela 5.1 apresenta os operadores de *pointcuts* primitivos, com uma breve descrição de cada um deles.

Operador	**Descrição**
call	Seleciona as chamadas de funções dentro de um módulo.
execution	Seleciona a execução de funções dentro de um módulo.
args	Seleciona *join points* dentro de um módulo baseado no número e padrões dos argumentos destes *join points*.
within	Seleciona os *join points* da definição de uma função. Os padrões da equação devem casar com o número e os padrões especificados pelo operador.

Tabela 5.1. Operadores primitivos de pointcut

[2] Tais regras estão relatadas em [Jon03].

5.3.2 Operadores para composição de pointcuts

É possível combinar os operadores primitivos para definição de *pointcuts* mais complexos. Existem três operadores para definição de *pointcuts* compostos: o operador unário ! e os operadores binários && e ||.

O operador unário !, quando aplicado a um *pointcut*, seleciona todos os *join points* não selecionados por seu operando. Este operador pode ser entendido como uma forma de obter o complemento do conjunto de *join points* selecionado por um *pointcut*. Como exemplo, a seguir apresentamos um *pointcut* que seleciona todos os *join points* de um programa, exceto as chamadas feitas a função *functionName* no módulo *Module*:

```
!call Module.functionName
```

O operador binário && seleciona apenas os *join points* selecionados por ambos os seus operandos. Ele pode ser visto como a operação de intersecção entre os conjuntos de *join points* selecionados por seus dois operandos. A seguir temos um exemplo de sua utilização:

```
execution Module.functionName &&
args id1 id2
```

Este *pointcut* seleciona os *join points* selecionados por *execution* e *args*, que são as equações da função *functionName* do módulo *Module* que possuam dois parâmetros formais (que serão ligados aos operadores *id1* e *id2*). Este caso representa uma combinação bastante freqüente, uma vez que a seleção de um *join point* é usualmente acompanhada da utilização de suas informações de contexto.

|| é um operador binário que seleciona os *join points* selecionados por seus dois operandos. Ele realiza a união entre os conjuntos de *join points* representados por seus operandos. Vejamos a seguir um exemplo:

```
execution Module.functionName ||
execution Module.functionName2
```

Os *join points* selecionados por este *pointcut* são as equações das funções *functionName* e *functionName2* do módulo *Module*.

É permitida qualquer combinação entre estes três operadores e os operadores primitivos. Uma combinação interessante, apresentada a seguir, é a que compõe um *pointcut* que seleciona todas as chamadas a uma função, com exceção das suas chamadas recursivas:

```
call Module.functionName &&
!within Module.functionName _ _
```

Os operadores && e || associam à esquerda e possuem a mesma precedência que é inferior à do operador !. Por exemplo,

```
! call Module.functionName ||
  args id1 id2
```

será interpretado como

```
(! call Module.functionName) ||
  args id1 id2
```

e não como

```
! (call Module.functionName ||
   args id1 id2)
```

A Tabela 5.2 apresenta breves descrições dos operadores para composição de *pointcuts* vistos nesta seção.

Operador	Descrição
!	Seleciona os *join points* não selecionados pelo seu operando.
&&	Seleciona os *join points* selecionados por ambos os operandos.
||	Seleciona todos os *join points* selecionados por seus operandos.

Tabela 5.2. Operadores para composição de pointcuts

5.4 DECLARAÇÕES DE ADVICE

AspectH permite definir computações adicionais que executam em *join points* específicos. A declaração de *advice* é a maneira pela qual tal especificação é feita. Uma declaração de *advice* deve indicar os *join points* de interesse (através de seu *pointcut*) e a computação que irá "decorar" de uma maneira determinada estes *join points*.

Não é permitida a definição de declarações de *advice* em módulos convencionais, apenas em aspectos. De maneira geral, as declarações de *advice* de um aspecto expressam, em conjunto, a implementação de um *crosscutting concern*. Observa-se também que é freqüente a existência de mais de uma declaração de *advice* de um mesmo aspecto, ou de aspectos diferentes, atuando em um mesmo *join point*. A linguagem, nesta situação, possui um critério definido para realizar a ordenação na execução destas declarações (Seção 5.6).

A declaração de *advice* é constituída de três partes:

- a maneira pela qual a computação será instalada no *join point*, isto é, o tipo do *advice*;
- o *pointcut*, que indica os *join points* que serão modificados e;
- a computação que será adicionada.

Abaixo temos um exemplo de declaração de *advice*:

```
before: execution Module.functionName
   = putStrLn "Before functionName..."
```

O primeiro componente desta declaração, antes do caractere *:*, é o tipo do *advice*, que no caso é *before*; a seguir temos o *pointcut* e, por fim, após o caractere =, temos a computação que executa nos *join points* selecionados.

A computação especificada em uma declaração de *advice* é uma ação monádica, que por sua vez são expressões que se deseja adicionar em um código monádico. Ao discutirmos os trabalhos relacionados (Seção 7.2), veremos que existe outra abordagem dedicada a integração de ações monádicas (Seção 7.2.5). Um exemplo são linguagens baseadas em regras de reescrita de símbolos. Lá veremos que AspectH é uma alternativa razoável a tais linguagens, oferecendo maior expressividade no que se refere a adição destas ações.

Uma restrição referente a computação especificada em uma declaração de *advice* é que esta possua tipo instância da classe *Monad*. Além disso, o tipo desta expressão deve ser compatível com o tipo de retorno dos *join points* selecionados, isto é, deve ser referente a mesma mônada. Questões relacionadas a tipos são discutidas na Seção 6.1.5.

5.4.1 Tipos de advice

Declarações de *advice* definem sua funcionalidade pela associação de uma computação a *join points*, e uma ordem, relativa a cada *join point* no *pointcut*, em que esta computação será executada. Esta ordem define o tipo do *advice*. Existem três tipos de *advice*, a saber, *before*, *after* e *around*, que indicam maneiras distintas de instalar a expressão do *advice* nos *join points* de interesse.

Um *advice* do tipo *before* indica que a expressão do *advice* será instalada antes dos *join points* selecionados. O *advice* apresentado acima, nesta seção, é do tipo *before* e define que a expressão *putStrLn "Before functionName..."* executará antes da execução da função *functionName* do módulo *Module*. Este *advice* é responsável pela impressão de uma mensagem antes da execução da função *functionName* e pode ser utilizado em um aspecto que capture a funcionalidade de *tracing* de um sistema.

O *advice* a seguir

```
after: execution Module.functionName
   = putStrLn "After functionName..."
```

é do tipo *after*. Como no *advice* apresentado anteriormente, este também imprime uma mensagem, mas desta vez, após a execução da função *functionName*.

Desta forma, declarações de *advice* do tipo *before* e *after* têm unicamente efeito aditivo: o *join point* selecionado executará de alguma maneira juntamente com a computação do *advice*.

Declarações de *advice* do tipo *around* indicam que a expressão do *advice* executará "ao invés" da computação dos *join points* interceptados, isto é, a computação definida no *advice* substituirá a execução do código em tais *join points*. No entanto, é possível reinstalar a computação interceptada através de uma chamada a função *proceed*, que representa individualmente os *join points* selecionados. Assim, a função *proceed* é útil

para inspecionar o valor retornado pelas funções interceptadas. A função *proceed* pode ser chamada livremente dentro do escopo da expressão do *advice*, devendo conter argumentos convenientes e compatíveis com relação aos *join points* que ela representa.

Com um *advice* do tipo *around*, é possível implementar os do tipo *before* e *after*. Vejamos a seguir os exemplos vistos anteriormente empregando *advice* do tipo *around*:

```
around: execution Module.functionName &&
                  args id1 id2
   = putStrLn "Before functionName..." >>
     proceed id1 id2

around: execution Module.functionName &&
                  args id1 id2
   = proceed id1 id2 >>
     putStrLn "After functionName..."
```

Utilizando a notação *do* temos o mesmo efeito:

```
around: execution Module.functionName &&
                  args id1 id2
   = do putStrLn "Before functionName..."
        proceed id1 id2

around: execution Module.functionName &&
                  args id1 id2
   = do proceed id1 id2
        putStrLn "After functionName..."
```

Observe que é necessário a exposição das informações de contexto da função *functionName* (por meio do operador *args*), diferentemente das declarações de *advice* anteriores. Isto se deve à necessidade de chamar a função *proceed* com os argumentos adequados.

Declarações de *advice* do tipo *around* oferecem diversas possibilidades para expressar computações que, por exemplo, executam condicionalmente os *join points* interceptados ou que os executam utilizando argumentos modificados; neste último caso, é necessário apenas indicar valores diferentes como argumento na chamada da função *proceed*.

5.4.2 Parâmetros de pointcut

Na Seção 5.3, vimos que os operadores de *pointcut args* e *within* expõem informações de contexto dos *join points* interceptados por eles. Isto é conseguido através da ligação dos identificadores nos padrões com os argumentos dos *join points* em questão.

Com os parâmetros de *pointcut* é possível definir declarações de *advice* mais sofisticadas. Como exemplo, no *advice* do tipo *before* apresentado no início desta seção, podemos exibir também o valor dos argumentos da função *functionName*:

```
before: execution Module.functionName &&
        args id1 id2
  = putStrLn ("Before functionName " ++ (show id1) ++ " " ++ (show id2))
```

Aqui, a função *show* apresenta os valores contidos em *id1* e *id2* como uma *String*.

Nesta seção já vimos outros exemplos do emprego de parâmetros de *pointcut*: as chamadas para a função *proceed* nas declarações de *advice* do tipo *around* acima possuem os argumentos expostos pelo operador *args*.

5.5 ASPECTOS

Os aspectos são as unidades modulares de AspectH. Por meio de um aspecto é possível implementar um *crosscutting concern* específico, que de outra maneira estaria espalhado no código base definido em módulos convencionais.

Assim, um sistema implementado em AspectH possui diversos aspectos, cada um dos quais responsável pela implementação separada de um *crosscutting concern*, além do código base, constituído de módulos Haskell. O interessante aqui, é que os detalhes referentes às interações entre aspectos e a combinação dos aspectos com o código base ficam a cargo do *Weaver*.

Em AspectH, um aspecto é uma entidade similar a um módulo de Haskell. De fato, a declaração de um aspecto é bastante semelhante a de um módulo e é feita através da palavra-chave *aspect*, em vez de *module*. Além disso, em um aspecto, é possível fazer uso de todas as declarações permitidas em Haskell, que são empregadas para auxiliar a implementação do aspecto. Contudo, as abstrações mais comumente encontradas em aspectos são as declarações de *advice* que contêm as informações necessárias para o processo de combinação: é apenas permitida a definição de declarações de *advice* em aspectos; utilizar tais declarações em módulos convencionais constitui um erro, que interrompe o processo de combinação.

Declarações de tipo, função e outras, feitas em um aspecto não podem ser utilizadas diretamente por outros aspectos ou módulos. Explicando de outra maneira, é proibida a especificação de aspectos em declarações de *import* de um módulo ou de outro aspecto. Entretanto, como veremos em mais detalhes posteriormente, esta é uma restrição apenas aparente, uma vez que as declarações de *advice*, que certamente referenciam as declarações feitas localmente no aspecto, modificam a semântica de outros módulos e aspectos. O emprego destas declarações, portanto, é feito de maneira indireta. Faz parte do processo de combinação (a cargo do *Weaver*) introduzir de maneira adequada os aspectos nas declarações de *import* dos módulos modificados por eles.

Em um sistema AspectH, é comum o cenário onde um mesmo *join point* é modificado por mais de um aspecto. Em determinadas situações é interessante ter o controle da ordem de execução de tais aspectos, onde é necessário determinar explicitamente qual aspecto tem precedência. Isto é feito através da cláusula *dominates*. Uma declaração *dominates* de um aspecto contém a lista dos aspectos que o aspecto em questão terá precedência durante o processo de combinação. Maiores detalhes sobre a precedência durante o processo de combinação são apresentados na Seção 5.6.

Como exemplo de definição de aspectos em AspectH, vejamos a seguir o aspecto *A1*:

```
1. aspect A1 where
2.
3. dominates A2
4.
5. before: execution Module.functionName
6.     = putStrLn ("Before functionName")
7.
8. after: execution Module.functionName
9.     = putStrLn ("After functionName")
```

O aspecto possui três declarações. A primeira utiliza a primitiva *dominates* (linha 2) para especificar que o aspecto *A1* terá precedência em relação ao aspecto *A2*. As duas outras declarações definem *advice* que executará antes e depois, respectivamente, da execução da função *functionName* do módulo *Module*.

5.6 ORDENAÇÃO DE ASPECTOS

Em geral, mais de uma declaração de *advice* pode se aplicar em um mesmo *join point*. Tais declarações podem estar definidas em aspectos diferentes ou em um mesmo aspecto. A ordem relativa de execução do código da declaração de *advice* deve ser bem definida. A ordenação é baseada no fato de que aspectos são as unidades primárias de *crosscutting concerns*. Desta forma, tal precedência é resolvida pela precedência relativa do aspecto em que a declaração de *advice* é feita.

A ordem de execução do código de declarações de *advice* é determinada da seguinte maneira:

- Quando duas declarações estão localizadas no mesmo aspecto, e são exatamente do mesmo tipo, executará primeiro aquela que foi definida primeiramente no aspecto, isto é, a que aparece antes no corpo do mesmo.

  Caso sejam de tipos diferentes, terá precedência o *advice* do tipo *before*; posteriormente o do tipo *around*; por fim, do tipo *after*. Desta forma, considerando que existem várias declarações de *advice* dos três tipos atuando em um mesmo *join point*, o *aspect weaver* seleciona primeiramente todas as do tipo *before*, depois do tipo *around* e então do tipo *after*.

  Esta regra existe porque um mesmo aspecto pode precisar definir várias declarações de *advice* que se aplicam em um mesmo *join point*. Isto ocorre com freqüência com *advice* do tipo *before* ou *after*, mas pode ocorrer com duas declarações do mesmo tipo.

  Como exemplo, vejamos o aspecto a seguir, onde as declarações de *advice* não estão completamente definidas; considere todas atuando em um mesmo *join point*:

  ```
  1.  aspect A where
  2.
  3.  after: ...
  ```

```
4.
5.  before: ...
6.
7.  before: ...
8.
9.  around: ...
10.
11. after: ...
12.
13. around: ...
```

A ordem de execução é a seguinte: primeiro executa as declarações feitas nas linhas 5 e 7, do tipo *before*, depois as definidas nas linhas 9 e 13, do tipo *around* e, em seguida, as do tipo *after*, definidas nas linhas 3 e 11.

- Consideremos o caso em que duas declarações de *advice* são feitas em dois aspectos diferentes, *A1* e *A2*. Nesta situação, se o aspecto A1 contiver em sua cláusula *dominates* o aspecto *A2*, a declaração de *advice* feita no aspecto *A1* terá precedência em relação a feita no *A2*. Vejamos um exemplo que ilustra esta situação:

```
aspect A1 where

dominates A2

after: ...

aspect A2 where

before: ...
```

Neste caso, o *advice after* do aspecto *A1* executará antes do *advice before* do aspecto *A2*.

- Em todos os outros casos, a ordem relativa entre duas declarações de *advice* é indefinida. Estes são os casos mais comuns – onde dois aspectos conceitualmente independentes definem declarações de *advice* que influenciam um mesmo *join point* – e não é preciso que o programador controle a ordenação relativa entre declarações de *advice*.

5.7 PROCESSO DE COMBINAÇÃO

O processo de combinação do código base Haskell com os aspectos envolve diversas etapas, que garantem a execução do código dos aspectos nos *join points* afetados. O código monádico contido nos módulos Haskell (bem como em aspectos) sofre transformações referentes à introdução apropriada das computações definidas em declarações

de *advice* e os próprios aspectos, ao final do processo, são transformados em módulos convencionais. Em resumo, as atividades que constituem o processo de combinação são as seguintes:

- ordenação de aspectos e declarações de *advice*;
- transformação de código monádico nos *join points* selecionados;
- inclusão de declarações de *import* nos módulos convencionais e aspectos; e
- a transformação de aspectos em módulos convencionais.

Nas próximas subseções apresentamos descrições de cada uma destas etapas.

5.7.1 Ordenação

A primeira etapa do processo de combinação é a definição de uma ordem para execução do código definido em aspectos, uma vez que, como vimos, é possível a execução simultânea de mais de uma declaração de *advice* (possivelmente definidas em aspectos diferentes) em um mesmo *join point*. A ordenação é referente tanto a aspectos quanto a declarações de *advice* definidas em um mesmo aspecto.

A definição de precedência entre aspectos é baseada nas listas de aspectos declaradas na cláusula *dominates*, como discutido na Seção 5.6. Caso tal informação não esteja disponível, a ordem de execução é indefinida, sendo resolvida internamente.

A ordem de execução do código definido em declarações de *advice* em um mesmo aspecto segue os critérios discutidos na Seção 5.6.

5.7.2 Transformação de código monádico

Esta etapa é responsável pela combinação do código definido em declarações de *advice* com os *join points* selecionados por elas. Como já mencionado anteriormente, os *join points* afetados podem estar localizados tanto em módulos convencionais como em aspectos.

A combinação se dá através de uma série de transformações de código que são escolhidas com base em diversos critérios que incluem o tipo de *join point* interceptado, o tipo de *advice* em questão e os operadores de *pointcut* empregados.

Na Seção 6.1 apresentamos os detalhes referentes a implementação da linguagem que incluem mais informações a respeito destas transformações.

5.7.3 Inclusão de declarações de import

A definição de um aspecto pode incluir opcionalmente declarações internas que auxiliam na implementação do código dos *crosscutting concerns*. Como este código é "transferido" para outras localidades (*join points*, definidos em aspectos ou módulos) é necessário que as declarações internas dos aspectos sejam visíveis de algum modo em tais localidades.

Para tanto o processo de combinação deve incluir apropriadamente os aspectos correspondentes nas listas de *import* dos módulos (ou aspectos) afetados por eles. Como veremos a seguir, os aspectos, ao fim do processo, são transformados em módulos Haskell; assim, suas declarações internas podem ser utilizadas por outros módulos.

Contudo, não é permitida a inclusão explícita pelo programador de aspectos nas listas de *import* de módulos ou aspectos. A utilização das definições de aspectos é feita indiretamente e é resolvida de forma automática pelo pré-processador.

5.7.4 Transformação de aspectos em módulos convencionais

A última etapa do processo de combinação transforma os aspectos do sistema em módulos convencionais. Isto porque, como mencionado, as definições internas de um aspecto são usadas remotamente nos módulos e aspectos modificados por ele. Esta transformação é realizada por três atividades simples:

- Troca da palavra-chave *aspect* por *module*;
- remoção de todas as declarações *dominates* e
- a remoção de todas as declarações de *advice*.

Ao final, o que se tem é um módulo contendo apenas declarações permitidas em Haskell e que pode ser importado como de costume.

5.8 ASPECTH ATRAVÉS DE UM EXEMPLO

Nesta seção apresentamos através de um pequeno exemplo a implementação de *crosscutting concerns* de um programa Haskell monádico utilizando a linguagem AspectH. Um dos objetivos aqui é dar uma visão concreta do emprego das construções disponibilizadas pela linguagem de maneira apropriada, ressaltando as vantagens e também as limitações da linguagem.

A princípio apresentamos uma visão geral da aplicação, que contém informações sobre seu propósito, detalhes relevantes para o entendimento do código, bem como uma descrição da organização modular do sistema. A seguir, temos explicações referentes aos tipos de dados utilizados e ao mecanismo de persistência de dados do sistema. As três subseções seguintes são dedicadas a implementação dos módulos e aspectos: os módulos convencionais realizam o código da funcionalidade básica da aplicação, enquanto que o código dos *crosscutting concerns* está localizado em aspectos. Apresentamos duas versões alternativas para implementação dos aspectos, que abrangem todos os operadores de *pointcut* vistos na Seção 5.3. Por fim, discutimos os pontos positivos e negativos da abordagem e conveniência de cada uma das versões dos aspectos apresentadas.

5.8.1 Características gerais da aplicação

O exemplo utilizado é uma aplicação que implementa as funcionalidades de uma biblioteca. A aplicação gerencia informações de usuários, livros, autores e editoras. Para

ter acesso ao sistema o usuário deve fornecer um *login* e uma senha que são autenticados pelo sistema.

Os usuários que fazem uso do sistema de biblioteca são classificados de acordo com seus direitos de acesso: usuários com status *ADMIN* possuem acesso privilegiado ao sistema, podendo manipular diretamente o banco de dados de usuários, livros, autores e editoras, sendo autorizados a realizar empréstimo e devolução de livros; usuários com status *USER* podem apenas obter informações sobre livros, autores e editoras.

A funcionalidade básica do sistema, referente a inclusão, remoção e alteração de dados das entidades, empréstimo e devolução de livros, bem como a interface com o usuário está organizado em dois módulos, a saber, *Library* e *Interface*.

O módulo *Library* contém os tipos de dados que modelam as entidades e as funções que manipulam diretamente o banco de dados do sistema. Estas funções realizam operações de *I/O* e, como vimos na Seção 4.1.5, todas as interações entre programas Haskell e o mundo exterior são realizadas por meio da mônada *IO*.

O módulo *Interface* possui as funções que implementam uma interface em modo texto com o usuário. As funções deste módulo fazem chamadas às funções do módulo *Library* para realizar as operações desejadas.

O sistema deve implementar determinadas verificações antes de realizar suas operações. Exemplos incluem a verificação de direitos de acesso do usuário, disponibilidade de cópias para empréstimo de um livro ou se o usuário já possui uma cópia do livro que pretende tomar emprestado. Estas verificações abrangem diversas funções do programa, sendo consideradas, portanto, *crosscutting concerns* da aplicação. Outras funcionalidades, também consideradas *crosscutting concerns*, são referentes ao controle de exceções do sistema e *tracing* de operações realizadas. Estes são exemplos clássicos de *crosscutting concerns* apresentados na literatura.

Os *crosscutting concerns* da aplicação são implementados de forma modular em aspectos. As duas versões do sistema possuem quatro aspectos: *ExceptionHandlingAspect*, *AuthenticationAspect*, *VerificationAspect* e *TracingAspect*. Os aspectos contêm definições internas e declarações de *advice* que afetam as funções do programa base. O que diferencia cada uma das versões é onde a interceptação do código é feita, isto é, nos tipos de *join points* selecionados. No caso da primeira versão, a interceptação do código é feita nas definições de funções do módulo *Library*: os *join points* selecionados são equações de funções. A segunda versão, por sua vez, modifica o código do módulo *Interface*, onde são feitas as chamadas das funções do módulo *Library*. Ao final da apresentação o leitor terá um entendimento mais consistente das diferenças entre os dois tipos de *join points* permitidos e de como é feita a especificação deles por meio dos operadores de *pointcut*.

5.8.2 Tipos de dados do sistema

Existem cinco entidades distintas presentes no sistema: usuários, livros, autores e editoras. Estas entidades são modeladas por meio de tipos algébricos de dados (Seção 3.1.3) localizados no módulo *Library*. A seguir temos os tipos de dados da aplicação:

```
data User = User String          -- login
                 String          -- senha
```

```
                String          -- direitos (ADMIN/USER)
                String          -- nome
                String          -- cpf

data Book = Book String         -- isbn
                Publisher       -- editora
                [Author]        -- autores
                String          -- título
                Int             -- disponíveis

data Author = Author String     -- cpf
                     String     -- nome

data Publisher = Publisher String -- cgc
                           String -- descrição
```

A descrição dos campos constituintes de cada entidade é exibida à direita, como comentário.

5.8.3 Mecanismo de persistência de dados

O mecanismo de persistência dos dados do sistema é implementado através de um banco de dados relacional. O acesso ao banco de dados é feito utilizando a biblioteca HaskellDB [BH04]. HaskellDB é uma biblioteca de combinadores para expressar consultas e outras operações em bancos de dados relacionais de forma declarativa e com segurança de tipos. Uma vez que as funções de HaskellDB realizam interações de *I/O*, as mesmas utilizam a mônada *IO* como *framework*. Todas as consultas e operações são completamente expressas dentro de Haskell: são desnecessários comandos embutidos (que utilizam SQL).

HaskellDB possui suporte a diversos bancos de dados. O banco de dados utilizado neste exemplo foi *SQLite* [Tea04b], mas é possível empregar qualquer um dos outros bancos suportados por HaskellDB, bastando apenas especificar uma função de conexão diferente. As consultas e operações, entretanto, são escritas de maneira independente do banco de dados empregado. As opções de acesso ao banco de dados e a função de conexão empregadas neste exemplo são apresentadas a seguir:

```
opts = SQLiteOptions {filepath="/home/andreazza/temp/haskell/library"
                     , mode=ReadWriteMode }
withDB :: (Database -> IO a) -> IO a
withDB = sqliteConnect opts
```

A função *withDB* deve receber como argumento uma função que recebe (uma referência para) o banco de dados e realiza uma operação de *IO*.

As informações sobre as tabelas do banco de dados, tais como nome e tipo dos campos, utilizadas nas consultas ficam contidas em módulos Haskell. Existe um módulo para cada tabela no banco. Estes módulos são criados a partir do *layout* do banco de

dados automaticamente por uma ferramenta disponibilizada por HaskellDB. Os módulos criados devem ser evidentemente importados pela aplicação que pretende empregá-los.

O banco de dados do sistema possui seis tabelas referentes às entidades do sistema, aos empréstimos e a relacionamentos entre as tabelas. Os módulos gerados, correspondentes a cada uma destas tabelas, são importadas pelo módulo *Library*:

```
import LibraryDB.Users as U       -- usuários
import LibraryDB.Books as B       -- livros
import LibraryDB.Authors as A     -- autores
import LibraryDB.Publishers as P  -- editoras
import LibraryDB.Borrows as BW    -- empréstimos
import LibraryDB.Bk_au as BA      -- relacionamento livro-autor
```

Observe a especificação de *aliases* para facilitar a referência às tabelas nas consultas. O módulo *LibraryDB.Authors*, por exemplo, é referenciado apenas como *A*.

5.8.4 Módulo Library

Como mencionado, o código base da aplicação está organizado nos módulos *Library* e *Interface*.

O módulo *Library* contém as funções que realizam as operações básicas, correspondentes a manipulação das entidades do sistema e a empréstimo e devolução de livros. Como exemplo, analisemos a seguir a função *addUser* que adiciona um usuário ao sistema:

```
1. addUser :: String -> User -> IO String
2. addUser me (User lg passwd rgts cpf nm) =
3.    withDB $ \db ->
4.    do insert db U.users (U.login    << constant lg #
5.                          U.password << constant passwd #
6.                          U.rights   << constant rgts #
7.                          U.cpf      << constant cpf #
8.                          U.name     << constant nm
9.                         )
10.      return "Ok"
```

A função *addUser* é definida por uma chamada à função de conexão *withDB* na linha 3 [3]. O argumento de *withDB* é uma abstração lambda que possui o argumento *db* que é utilizado como referência ao banco de dados no lado direito da abstração lambda. A inserção propriamente dita é realizada pela função *insert* de HaskellDB (linha 4). Os dois primeiros argumentos da função são o banco de dados (*db*) e a tabela onde será feita a inserção (*U.users*). O último argumento especifica os valores que serão inseridos nos campos da tabela: o operador << relaciona o nome do campo com o valor que será

[3] O operador $ aplica uma função a seus argumentos; ele é empregado para evitar o uso excessivo de parêntesis.

inserido nele [4]. Ao fim da operação a função *addUser* retorna a *String* "ok", indicando que a operação foi realizada com sucesso.

As funções responsáveis pela remoção e atualização dos dados de um usuário são apresentadas abaixo:

```
1.  rmUser :: String -> String -> IO String
2.  rmUser me lg =
3.    withDB $ \db ->
4.    do delete db U.users (\r -> r!U.login .==. constant lg)
5.       return "ok"
6.
7.  updUser :: String -> User -> IO String
8.  updUser me (User lg passwd rgts cpf nm) =
9.    withDB $ \db ->
10.   do update db U.users
11.         (\r -> r!U.login .==. constant lg)
12.         (\r -> U.password << constant passwd #
13.                U.rights   << constant rgts #
14.                U.name     << constant cpf #
15.                U.cpf      << constant nm)
16.      return "ok"
```

O código destas funções é similar ao de *addUser*. As funções *delete* (linha 4) e *update* (linha 10) de HaskellDB são responsáveis, respectivamente, por remover e atualizar registros de uma tabela.

As funções referentes às demais entidades do sistema – livros, autores e editoras – são definidas de maneira semelhante e, por este motivo, apresentamos apenas suas declarações de tipo que serão importantes para o entendimento do código dos aspectos:

```
addBook :: String -> Book -> IO String
rmBook :: String -> String -> IO String
updBook :: String -> Book -> IO String

addAuthor :: String -> Author -> IO String
rmAuthor :: String -> String -> IO String
updAuthor :: String -> Author -> IO String

addPublisher :: String -> Publisher -> IO String
rmPublisher :: String -> String -> IO String
updPublisher :: String -> Publisher -> IO String
```

A operação responsável pelo empréstimo de livros (vista a seguir) deve realizar duas operações de maneira atômica: inserir o registro correspondente na tabela de

[4] A função *constant* converte o valor a ser inserido em um tipo interno que representa constantes em HaskellDB.

empréstimos (linhas 5 a 7) e atualizar a tabela de livros referente a diminuição de exemplares disponíveis (linhas 8 a 11):

```
1. borrow :: String -> String -> String -> IO String
2. borrow me isbn cpf =
3.   withDB $ \db ->
4.   transaction db $
5.   do insert db BW.borrows (BW.isbn << constant isbn #
6.                            BW.cpf  << constant cpf
7.                           )
8.      (Book _ _ _ _ avl) <- getBook isbn
9.      update db B.books
10.            (\r -> r!B.isbn .==. c isbn)
11.            (\r -> B.available << c (avl-1))
12.     return "ok"
```

Para garantir que as operações sejam realizadas atomicamente, HaskellDB disponibiliza a função *transaction* (linha 4), usada no código acima.

A seguir temos a função que realiza a devolução de livros. Ela deve apagar o registro da tabela de empréstimos (linhas 5 a 7) e atualizar o número de exemplares disponíveis na tabela de livros (linhas 8 a 11) de maneira atômica (linha 4):

```
1. deliver :: String -> String -> String -> IO String
2. deliver me isbn cpf =
3.     withDB $ \db ->
4.     transaction db $
5.     do delete db BW.borrows
6.          (\r -> r!BW.cpf .==. constant cpf .&&.
7.                 r!BW.isbn .==. constant isbn)
8.        (Book _ _ _ _ avl) <- getBook isbn
9.        update db B.books
10.             (\r -> r!B.isbn .==. constant isbn)
11.             (\r -> B.available << constant (avl+1))
12.       return "ok"
```

5.8.5 Módulo Interface

O módulo *Interface* é responsável por uma interface em modo texto com o usuário. As funções deste módulo coletam os dados do usuário e fazem chamadas a funções de *Library*. Como exemplo, analisemos a seguir a função *insertUser*:

```
1. insertUser :: String -> IO ()
2. insertUser me =
3.   do putStr "login:"
4.      lg <- getLine
5.      putStr "password:"
```

```
6.      pw <- getLine
7.      putStr "rights(ADMIN/USER):"
8.      rgts <- getLine
9.      putStr "name:"
10.     nm <- getLine
11.     putStr "cpf:"
12.     cpf <- getLine
13.     res <- addUser me (User lg pw rgts nm cpf)
14.     putStrLn res
```

Esta função obtém os dados necessários para inclusão de um usuário no sistema, ligando-os aos identificadores *lg*, *pw*, *rgts*, *nm* e *cpf*. Em seguida, a função *addUser* é chamada (linha 13); o valor retornado pela função é ligado ao identificador *res* que é, então, impresso na saída padrão (linha 14).

As demais funções operam de modo semelhante; por este motivo apresentamos a seguir apenas suas definições de tipo:

```
updateUser :: String -> IO ()
removeUser :: String -> IO ()

insertBook :: String -> IO ()
updateBook :: String -> IO ()
removeBook :: String -> IO ()

insertAuthor :: String -> IO ()
updateAuthor :: String -> IO ()
removeAuthor :: String -> IO ()

insertPublisher :: String -> IO ()
updatePublisher :: String -> IO ()
removePublisher :: String -> IO ()

borrowBook :: String -> IO ()
deliverBook :: String -> IO ()
```

5.8.6 Aspectos: primeira versão

O sistema de biblioteca, como apresentado até aqui nos módulos *Library* e *Interface*, é capaz de realizar as operações básicas esperadas. Entretanto, os requisitos que respondem pelos *crosscutting concerns* estão ausentes. A implementação de tais requisitos envolve a existência de código que influencia de maneira idêntica a semântica de grupos de funções no código base.

Os *crosscutting concerns* do sistema dizem respeito especificamente ao controle de exceções, autenticação de operações baseada nos direitos de acesso do usuário, verificações necessárias ao empréstimo de livros e *tracing* de operações para depuração do

código do sistema. Apresentamos aqui a primeira versão dos aspectos, que modificam as funções do módulo *Library*, selecionando diretamente a execução de suas funções.

Comecemos pelo controle de exceções. As funções desta aplicação utilizam a mônada *IO* para realizar interação com o mundo exterior. O que fazer, então, quando ocorre uma falha que previna a conclusão de uma operação de *I/O*, como, por exemplo, falhas na comunicação com o banco de dados? *IO* disponibiliza a função *catch* que especifica uma interação de *I/O* juntamente com uma função que executa sempre que acontecer uma falha nesta interação:

```
catch :: IO a -> (IOError -> IO a) -> IO a
```

Assim, o aspecto responsável pelo controle de exceções deve interceptar a execução das funções do módulo *Library*, que realizam interações de *I/O*, e especificar, em substituição, uma chamada à função *catch* que tenha como primeiro argumento os *join points* interceptados. O aspecto *ExceptionHandlingAspect*, que está a cargo do controle de exceções, é apresentado a seguir:

```
1.  aspect ExceptionHandlingAspect where
2.
3.  around: ( execution Library.addUser       ||
4.            execution Library.addBook       ||
5.            execution Library.addAuthor     ||
6.            execution Library.addPublisher  ||
7.            execution Library.updUser       ||
8.            execution Library.updBook       ||
9.            execution Library.updAuthor     ||
10.           execution Library.updPublisher  ||
11.            execution Library.rmUser        ||
12.           execution Library.rmBook        ||
13.           execution Library.rmAuthor      ||
14.           execution Library.rmPublisher ) &&
15.         args me entity
16.   = catch (proceed me entity)
17.           (\e -> return ("Error: "++(show e)))
18.
19. around: ( execution Library.borrow    ||
20.           execution Library.deliver ) &&
21.         args me isbn cpf
22.   = catch (proceed me isbn cpf)
23.           (\e -> return ("Error: "++(show e)))
```

Este aspecto possui duas declarações de *advice* do tipo *around*. A primeira delas intercepta a execução de funções que manipulam diretamente as entidades do sistema, selecionadas por meio do operador *execution*. Observe que as chamadas a este operador são compostas por meio do operador ||, significando que qualquer uma destas funções é

join point válido. A composição com o operador *args* (linha 15) por meio de && (linha 14), por sua vez, liga os identificadores *me* e *entity* aos argumentos do *join point* (todas as funções selecionadas pelo operador *execution* possuem dois argumentos). O *advice* especifica uma expressão que reinstala os *join points* interceptados por meio de uma chamada à função *proceed* (linha 16). Caso haja algum erro de *I/O*, a abstração lambda contida no segundo argumento de *catch* é executada, retornando uma mensagem com a descrição do erro ocorrido.

A outra declaração de *advice*, especifica uma computação bastante semelhante. Ela é necessária porque seus *join points* (as definições das funções *borrow* e *deliver*) são, por assim dizer, incompatíveis com os *join points* especificados pela primeira: possuem números de argumentos diferentes. Esta repetição desnecessária de código é resultado de uma limitação intrínseca da abordagem que atrela por demais o código dos aspectos à representa textual dos programas.

A autenticação das operações, no tocante aos direitos de acesso do usuário, é implementada pelo aspecto *AuthenticationAspect*:

```
1.  aspect AuthenticationAspect where
2.
3.  dominates ExceptionHandlingAspect, VerificationAspect
4.
5.  getUserRigths :: String -> IO String
6.  getUserRights lg =
8.      withDB $ \db ->
9.      do let q1 = do p <- table U.users
10.                    restrict (p!U.login .==. constant lg)
11.                    project (rights << p!rights)
12.        res <- query db q1
13.        let [rgt] = map (\r -> (r!rights)) res
14.        return rgt
15.
16. authenticateAdmin :: String -> IO String
17. authenticateAdmin lg
18.   = do rgt <- getUserRights lg
19.        let auth = case rgt of
20.                     ADMIN -> True
21.                     USER -> False
22.        return auth
23.
24. around: ( execution Library.add **  ||
25.           execution Library.upd **  ||
26.           execution Library.rm ** ) &&
27.         args me entity
28.   = withDB $ \db ->
29.     do auth <- authenticateAdmin me
30.        if auth
```

```
31.          then proceed me entity
32.          else return "You can't!!!"
33.
34. around: ( execution Library.borrow    ||
35.           execution Library.deliver ) &&
36.         args me isbn cpf
37.   = withDB $ \db ->
38.     do auth <- authenticateAdmin me
39.        if auth
40.          then proceed me isbn cpf
41.          else return "You can't!!!"
```

O aspecto possui duas funções que auxiliam a implementação de sua funcionalidade: *getUsersRights* e *authenticateAdmin*. A primeira delas obtém os direitos de acesso de um usuário, enquanto que a outra decide se o usuário possui status *ADMIN*.

As declarações de *advice* neste aspecto realizam a autenticação das funções interceptadas por eles, e são bastante semelhantes às definidas no aspecto *ExceptionHandlingAspect*, afetando exatamente os mesmos *join points*. Note, contudo, na primeira declaração de *advice*, o emprego de coringas: uma vez que existem grupos de funções cujos identificadores possuem prefixo em comum, é possível especificar um *pointcut* de maneira menos prolixa do que o visto no aspecto anterior. A computação definida nas duas declarações de *advice* realiza uma chamada à função *authenticateAdmin* (linhas 29 e 38). A execução dos *join points* interceptados é condicionada, então, ao valor calculado por esta função que é ligado ao identificador *auth*.

O aspecto *AuthenticationAspect* especifica, através da cláusula *dominates* (linha 3), que sua execução tem precedência em relação aos aspectos *ExceptionHandlingAspect* e *VerificationAspect*. Isto é necessário uma vez que suas declarações de *advice* chamam a função *authenticateAdmin*, que pode causar um erro de *I/O*. O código desde aspecto deve, portanto, ser combinado com o código base antes que o do aspecto que manipula exceções, para que esta possibilidade seja tratada adequadamente. A precedência em relação a *VerificationAspect* é necessária pelo fato de que, como veremos, as verificações realizadas neste aspecto são desnecessárias caso o usuário não tenha status *ADMIN*.

O aspecto *VerificationAspect* atua na função responsável pelo empréstimo de livros:

```
1.  aspect VerificationAspect where
2.
3.  dominates ExceptionHandlingAspect
4.
5.  hasAvailable :: String -> IO String
6.  hasAvailable isbn =
7.      withDB $ \db ->
8.      do let q1 = do p <- table B.books
9.                     restrict (p!B.isbn .==. constant isbn)
10.                    project (B.available << p!available)
11.        res <- query db q1
```

```
12.        let [avl] =  map (\r -> (r!available)) res
13.        return $ avl /= 0
14.
15. hasAlreadyACopy :: String -> String -> IO String
16. hasAlreadyACopy isbn cpf =
17.     withDB $ \db ->
18.     do let q1 = do p <- table BW.borrows
19.                    restrict (p!BW.cpf .==. constant cpf)
20.                    project (BW.isbn << p!BW.isbn)
21.        res <- query db q1
22.        let books = map (\r -> (r!BW.isbn)) res
23.        return $ books /= []
24.
25. around: execution Library.borrow &&
26.         args me isbn cpf
27.   = do av <- hasAvailable isbn
28.        al <- hasAlreadyACopy isbn cpf
29.        if av && (not al)
30.         then proceed me isbn cpf
31.         else return "I don't borrow!!!"
```

O aspecto realiza duas verificações necessárias ao prosseguimento desta operação. Uma delas é se existem exemplares do livro desejado disponíveis para empréstimo na biblioteca; a função *hasAvailable* realiza esta tarefa. A outra (*hasAlreadyACopy*) verifica se o usuário em questão já possui uma cópia do livro que deseja tomar emprestado. Caso já tenha, a função *hasAlreadyACopy* retorna o valor *True*. A única declaração de *advice* deste aspecto (linha 25 em diante) seleciona a função *borrow*; a execução desta função (através da chamada a *proceed*) é condicionada (linha 29) aos valores contidos em *av* (linha 27) e *al* (linha 28) que contêm os valores calculados por *hasAvailable* e *hasAlreadyACopy*.

A execução deste aspecto também possui precedência em relação ao aspecto *ExceptionHandlingAspect* (ver cláusula *dominates* na linha 3), já que é necessário tratar as possíveis falhas de *I/O* que seu código pode provocar.

Vejamos, por fim, o aspecto referente a funcionalidade de *tracing*, *TracingAspect*. Este aspecto monitora a execução de funções com o intuito de averiguar erros presentes no código. Este aspecto é necessário apenas para auxiliar a implentação, sendo descartado quando a aplicação está pronta. Vejamos a seguir o aspecto *TracingAspect*:

```
1. aspect TracingAspect where
2.
3. before: ( execution Library.addUser  ||
4.           execution Library.updUser  ||
5.           execution Library.rmUser ) &&
6.         args me (User lg pw rgts cpf nm)
7.    = putStrLn ("cpf:"++cpf++", nome:"++nm)
```

Sua declaração de *advice* seleciona as funções *addUser*, *updUser* e *rmUser* e imprime (linha 7) os valores contidos em *cpf* e *nm* referentes ao usuário que está sendo manipulado.

A implementação dos *crosscutting concerns* do sistema de biblioteca em aspectos oferece diversas vantagens, todas decorrentes do maior grau de modularidade provido. O código dos quatro aspectos apresentados, de outra forma, estaria misturado e espalhado pelas funções do módulo *Library*, dificultando tanto a manutenção do código dos *crosscutting concerns* quanto a legibilidade das funções do código base. É interessante analisar, então, como ficaria o código de uma função implementada da maneira tradicional sem aspectos para contrapor com a versão em AspectH. A seguir apresentamos uma versão da função *addUser* que implementa seus *crosscutting concerns* diretamente em sua definição:

```
1. addUser :: String -> User -> IO String
2. addUser me (User lg pw rths cpf nm) =
3.   catch
4.    (withDB $ \db ->
5.     do auth <- authenticateAdmin
6.        if auth
7.          then do putStrLn ("cpf:"++cpf++", nome:"++nm)
8.                  insert db users (U.login    << constant lg #
9.                                   U.password << constant passwd #
10.                                  U.rights   << constant rgt #
11.                                  U.cpf      << constant cpf #
12.                                  U.name     << constant nm
13.                                 )
14.                 return "ok"
15.         else return "You can't!!!")
16.    (\e -> return ("Error: " ++ (show e)))
```

Compare com o código de *addUser* visto na Seção 5.8.4. A legibilidade da função é comprometida, uma vez que ela realiza, juntamente com seu propósito básico, diversas funcionalidades não relacionadas. Ela ilustra a falta de modularidade resultante de uma abordagem que utiliza apenas os recursos tradicionais.

5.8.7 Aspectos: segunda versão

Os aspectos apresentados anteriormente modificam o código das funções do módulo *Library* diretamente nas equações que definem suas funções. É possível, entretanto, implementar os aspectos de modo a afetar o outro tipo de *join points* permitido (chamadas de funções) e conseguir o mesmo resultado, que é a separação do código dos *crosscutting concerns*.

A versão apresentada aqui atua no código do módulo *Interface*, onde as funções de *Library* são chamadas. A alteração no código dos aspectos ocorre, portanto, apenas no conjunto de *join points* selecionado por suas declarações de *advice*. Isto é interessante para um exame de todos os operadores de *pointcut* providos por AspectH.

Analisemos, então, como fica o aspecto *ExceptionHandlingAspect*:

```
1.  aspect ExceptionHandlingAspect where
2.
3.  around: ( (within Interface.insertUser _ &&
4.              call Interface.addUser) ||
5.            (within Interface.updateUser _ &&
6.              call Interface.updUser) ||
7.            (within Interface.removeUser _ &&
8.              call Interface.rmUser)  ||
9.            (within Interface.insertBook _ &&
10.             call Interface.addBook) ||
11.           (within Interface.updateBook _ &&
12.             call Interface.updBook) ||
13.           (within Interface.removeBook _ &&
14.             call Interface.rmBook)  ||
15.           (within Interface.insertAuthor _ &&
16.             call Interface.addAuthor) ||
17.           (within Interface.updateAuthor _ &&
18.             call Interface.updAuthor) ||
19.           (within Interface.removeAuthor _ &&
20.             call Interface.rmAuthor)  ||
21.           (within Interface.insertPublisher _ &&
22.             call Interface.addPublisher) ||
23.           (within Interface.updatePublisher _ &&
24.             call Interface.updPublisher) ||
25.           (within Interface.removePublisher _ &&
26.             call Interface.rmPublisher) )
27.         && args me entity
28.   = catch (proceed me entity)
29.           (\e -> return "Error: "++(show e))
30.
31. around: ( (within Interface.insertUser _ &&
32.             call Interface.borrow) ||
33.           (within Interface.updateUser _ &&
34.             call Interface.deliver) )
35.         && args me isbn cpf
36.   = catch (proceed me isbn cpf)
37.           (\e -> return "Error: "++(show e))
```

Os *pointcuts* são mais prolixos devido a identificação mais complicada dos *join points*, que são chamadas feitas no interior de definições de outras funções. A identificação dos *join points* é feita individualmente, não cabendo neste caso a utilização de coringas. O operador *within* especifica onde é feita a chamada da função; observe a utilização do padrão _, uma vez que a informação de contexto referente às definições de funções são desnecessárias neste exemplo. O operador *call* especifica a chamada de função que se pretende selecionar; note que o módulo referido neste operador é *Interface* onde a função

é chamada, e não *Library*, onde ocorre a definição. A composição dos operadores *within* e *call* através do operador binário && amarra a chamada especificada ao local onde ela é feita. O operador *args* (linhas 27 e 35) expõe a informação de contexto necessária para utilização nas expressões das declarações de *advice*.

O código dos demais aspectos será omitido, já que as mudanças realizadas se restringem aos *pointcuts*, que são bastante semelhantes ao do aspecto *ExceptionHandlingAspect*.

As modificações de código providas por esta versão proporcionam exatamente o mesmo comportamento externo para o sistema. A diferença existe apenas nos locais do código base afetado pelo código dos aspectos. Para apreciar tal diferença, vejamos a implementação dos *crosscutting concerns* do sistema, realizada na função *insertUser* do módulo *Interface*:

```
1. insertUser :: String -> IO ()
2. insertUser me =
3.   do putStr "login:"
4.      lg <- getLine
5.      putStr "password:"
6.      pw <- getLine
7.      putStr "rights(ADMIN/USER):"
8.      rgts <- getLine
9.      putStr "name:"
10.     nm <- getLine
11.     putStr "cpf:"
12.     cpf <- getLine
13.     res <- catch
14.             (do auth <- authenticateAdmin me
15.                 if auth
16.                   then do putStrLn $ "cpf:"++cpf++", nome:"++nm
17.                           addUser me (User lg pw rgts nm cpf)
18.                   else return "You can't!!!")
19.              (\e -> return ("Error: " ++ (show e)))
20.     putStrLn res
```

Note que, a despeito do local, a qualidade do código, no que diz respeito a modularidade, é inferior se comparado com a implementação realizada em aspectos: menos legível, mais difícil de manter e reusar.

5.8.8 Discussão

O programa apresentado nesta seção é bastante útil para o entendimento das distinções entre os dois tipos de *join point* permitidos, bem como ajuda a esclarecer de forma mais precisa como a especificação destes *join points* é feita. Outro lado positivo é a percepção de que, independente de onde é feita a composição, se na execução ou na chamada de uma função, o efeito externo resultante é equivalente.

A identificação de código feita na definição da função garante que o código dos aspectos que modificam sua semântica executa sempre, para todas as suas chamadas.

Contudo, pode ser interessante, e em algumas ocasiões inevitável, fazer tal identificação no local da chamada. Interessante porque é possível especificar comportamentos diferentes dependendo de onde a chamada é feita; e inevitável, quando, por exemplo, a função pertence a uma biblioteca de terceiros ou é definida internamente, em que não se deseja ou não se pode modificar.

O exemplo também presta-se a uma avaliação inicial das vantagens que a implementação de *crosscutting concerns* realizada em aspectos possui quando comparada com a maneira tradicional, feita diretamente no programa base. As funções do código base são mais legíveis, já que sua funcionalidade não está poluída com diversos outros interesses. Tanto o programa base como o dos aspectos tornam-se mais fáceis de manter e reusar. Além disso é possível "desplugar" livremente os aspectos tendo versões diferentes do sistema de forma instantânea.

No Capítulo 6 apresentamos uma discussão mais aprofundada e abrangente das vantagens e desvantagens da abordagem implementada por AspectH. Lá discutimos um estudo de caso que explora outros "aspectos", negligenciados neste exemplo.

CAPÍTULO 6

ESTUDO DE CASO

Implementações de Haskell disponibilizam diversas mônadas com amplo suporte, que auxiliam o programador em tarefas específicas, possibilitando a ele se concentrar diretamente no código que utiliza as definições monádicas e as facilidades providas. O programa descrito na Seção 5.8 é um exemplo deste cenário, onde a mônada *IO* de Haskell foi empregada para realizar as interações de *I/O* necessárias.

Existem situações, no entanto, onde é necessária a definição pelo programador de uma mônada para estruturar seu código, uma vez que as soluções já prontas não resolvem de maneira adequada seu problema. O *Weaver* de AspectH, apresentado neste capítulo, nos remete a este cenário. Sua implementação envolve a definição de uma mônada que mantém um ambiente (estado) que pode ser inspecionado e atualizado de forma livre, possuindo suporte a manipulação de exceções e a emissão de advertências.

Uma característica interessante do ciclo de vida deste programa foi que a mônada não esteve presente deste o início de seu desenvolvimento. Grande parte das funções do *Weaver* estava (parcialmente) pronta quando a mônada ainda nem havia sido definida. Quando a presença das funcionalidades impuras tornou-se inevitável, aí sim, a mônada foi implementada e todas as funções de alguma forma relacionadas com estas funcionalidades foram reestruturadas empregando as definições monádicas. Tal reestruturação incluiu a implementação de seus *crosscutting concerns* da maneira tradicional (juntamente com o código base), resultando nos efeitos negativos já discutidos em detalhes nos Capítulos 2 e 5.

Assim, surge a oportunidade de um estudo de caso interessante para AspectH, que é empregar aspectos na implementação de seu próprio *Weaver*. Desta forma, podemos avaliar de maneira mais ampla a linguagem, utilizando um programa de maiores dimensões e cujo suporte monádico fez parte do desenvolvimento, isto é, não se empregou definições já prontas.

Este capítulo está dedicado principalmente a descrever este estudo de caso. A princípio apresentamos as características principais do pré-processador de AspectH, detalhando os principais recursos utilizados, as transformações de código que implementam o processo de combinação dos aspectos com o código base e as questões de tipo relacionadas com o processo de combinação. A seguir discutimos detalhes acerca do *Weaver* e dos tipos de dados e funções envolvidos na definição da mônada, apresentando também os *crosscutting concerns* presentes em sua implementação (Seção 6.2). Então, tratamos dos aspectos do *Weaver* que implementam os *crosscutting concerns* do sistema de maneira separada do código base, garantindo um maior grau de modularidade no programa resultante (Seção 6.3). Apresentamos uma discussão que inclui os resultados conseguidos especificamente no estudo de caso e a linguagem AspectH de modo geral, com ênfase em suas vantagens e limitações, já vislumbrando direções possíveis para o futuro nesta linha de pesquisa (Seção 6.4). Por fim, na Seção 6.5, discorremos em torno da estratégia

adotada na implementação do *Weaver*, em que se evitou o emprego de mônadas desde o início do desenvolvimento, utilizando-se aspectos ao final para obter um resultado melhor no tocante à modularidade do programa.

6.1 O PRÉ-PROCESSADOR ASPECTH

Uma implementação AOP deve garantir que o código dos aspectos e o programa base executem conjuntamente de forma coordenada. O processo de combinação é responsável pela execução de uma declaração de *advice* nos *join points* apropriados. Como é o caso de outras funcionalidades de uma linguagem de programação, a combinação pode ser implementada por um pré-processador, durante a combinação, como parte de uma máquina virtual, entre outros.

A implementação da linguagem AspectH foi realizada por meio de um pré-processador fonte-a-fonte. A entrada para o processo de combinação são módulos Haskell e aspectos. Como saída, o pré-processador fornece apenas módulos Haskell que refletem as contribuições semânticas dos módulos e aspectos originais. Por utilizar um pré-processador, o processo de combinação é totalmente estático: todas as decisões acerca de execução do código de aspectos é feita estaticamente. Entretanto, a linguagem suporta quantificação estática e dinâmica, tendo em vista que as modificações realizadas possuem reflexos durante a execução, em resposta a eventos como chamadas de função, por exemplo.

O pré-processador de AspectH realiza a combinação de maneira puramente sintática, onde a identificação de *join points* (comparação de identificadores, casamento de padrões, etc.) é realizada com base na representação literal dos termos de um programa. Por exemplo, o *pointcut* abaixo:

```
args Div x y
```

identifica apenas a ocorrência exata do padrão *Div x y*; a forma alternativa (e equivalente), que usa o construtor como operador infixo, *x `Div` y*, não casa com o padrão definido no operador *args*.

O processo de combinação ocorre no nível da sintaxe abstrata: primeiramente é feita a análise sintática dos programas envolvidos no processo, após o que ocorre a transformação do código base, baseada nas regras de combinação definidas nos aspectos. Pelo fato de o *Weaver* trabalhar com árvores sintáticas, ele garante a inexistência de erros sintáticos nos programas resultantes. O *Weaver*, entretanto, não garante a corretude de tipos. Mais a frente nesta seção discutimos as questões referentes a tipos.

O restante desta seção descreve outros detalhes relevantes durante a implementação do pré-processador.

6.1.1 Criação das árvores sintáticas

AspectH é um superconjunto de Haskell, o que significa que programas válidos em Haskell também o são em AspectH. A linguagem apenas definiu novas construções como extensões sintáticas a Haskell.

Para a implementação do pré-processador fizemos uso de recursos fornecidos pelo projeto do compilador de Haskell GHC [Tea04a]. Este projeto disponibiliza facilidades para

realização de extensões sintáticas em Haskell. Tais facilidades incluem analisadores léxico e sintático, um modelo para sintaxe da linguagem (sintaxe abstrata) e um *pretty-printer* (que retorna a representação textual de uma árvore sintática). As extensões sintáticas são feitas por meio de modificações adequadas nestes recursos como, por exemplo, inclusão de novas palavras-chave no analisador léxico, novas regras e produções para o analisador sintático e alterações correspondentes na sintaxe abstrata. As modificações realizadas para implementação do pré-processador de AspectH permitem que seja feita a análise sintática tanto de módulos Haskell quanto de aspectos, não sendo necessário um novo analisador sintático específico para AspectH.

Para utilizar o pré-processador o usuário fornece os módulos Haskell e os aspectos, que são, então, transformados em suas representações em sintaxe abstrata. A lista de árvores sintáticas criadas pelo analisador sintático serve de entrada para o programa (*Weaver*) que realiza o processo de combinação propriamente dito.

6.1.2 Weaver

O processo de combinação fica a cargo de um programa em Haskell, o *Weaver*, que realiza as transformações de código necessárias para a execução apropriada do código dos aspectos nos *join points* escolhidos. Desta forma, cada *join point* de um programa é examinado de tal forma a decidir se há alguma declaração de *advice* que se aplica aí. Em caso afirmativo, este *join point* é modificado para refletir as mudanças descritas pela declaração de *advice*.

A entrada para o *Weaver* é uma lista de módulos e aspectos em sintaxe abstrata que se deseja combinar. Após a combinação do código dos aspectos com os módulos, o *Weaver* transforma os aspectos em módulos. A lista dos módulos originais e dos aspectos transformados em módulos é retornada como saída pelo *Weaver*, ainda como árvores sintáticas. Finalmente, o pré-processador transforma estas árvores sintáticas em suas representações textuais utilizando o *pretty-printer* fornecido pelo projeto GHC.

A implementação do *Weaver* abrange diversas funcionalidades impuras como estado entre computações e controle de exceções. Assim, uma mônada específica foi desenvolvida para lidar com tais funcionalidades. Esta mônada mantém um ambiente (estado) que armazena diversas informações pertinentes ao processo de combinação.

Cada módulo e aspecto da lista fornecida como entrada para o *Weaver* é examinado para decidir se seus *join points* são afetados por alguma declaração de *advice* e, então, transformado. O ambiente contém esta unidade modular presentemente examinada pelo *Weaver* e cada aspecto que pode eventualmente modificá-la, evitando que estas informações sejam passadas como argumento entre diversas funções. Outros dados presentes neste ambiente incluem um mapeamento entre cada aspecto e os *join points* que ele modifica. A mônada provê duas funções que manipulam diretamente o ambiente: uma é responsável por atualizar os valores armazenados no ambiente, enquanto que a outra retorna o estado atual contido no ambiente para inspeção.

A definição da mônada ainda inclui um mecanismo de controle de exceções, sendo também responsável pela emissão de advertências durante o processo de combinação.

Um subconjunto das funções que implementam o *Weaver* possuem como tipo de retorno a mônada descrita acima. O código monádico que as define "interage" com o

ambiente, examinando seu estado atual e atualizando seus valores.

A versão original do código do *Weaver* foi implementada apenas em Haskell. No que diz respeito a modularidade das funções monádicas, este código deixa a desejar uma vez que seus *crosscutting concerns* foram implementados juntamente com o código base, prejudicando a qualidade do programa resultante. Na Seção 6.2, apresentamos uma nova versão para este programa, que implementa apenas a funcionalidade básica do sistema. Seus *crosscutting concerns*, são implementados de maneira separada em aspectos, apresentados na Seção 6.3.

O código do *Weaver* está organizado em três módulos, que contêm as funções e as declarações de tipos responsáveis pelo processo de combinação:

- **WeaverMonad**. Este módulo contém os tipos de dados e as funções que definem a mônada do processo de combinação.
- **WeaverUtils**. Este módulo contém funções (não-monádicas) que auxiliam a implementação das funções do módulo *Weaver*.
- **Weaver**. Este módulo contém as principais funções do *Weaver*, encerrando a maior parte do código empregado em sua implementação. A maior parte de suas funções tem a mônada definida no módulo *WeaverMonad* como tipo de retorno.

6.1.3 Mecanismo de join points

A implementação pelo *Weaver* do mecanismo que decide se um *pointcut* seleciona um determinado *join point* foi bastante influenciada pelo trabalho descrito em [WKD02]. Lá os autores apresentam um modelo de *join points*, *pointcuts* e declarações de *advice* e introduzem uma pequena linguagem que incorpora estes recursos. No que se refere aos *pointcuts*, o artigo sugere uma função que possui dois argumentos (o *pointcut* e o *join point*) e realiza uma recursão estrutural nestes argumentos para decidir se o *join point* faz ou não parte do *pointcut*.

A sintaxe abstrata de AspectH inclui modelos para os operadores de *pointcut*, bem como para os dois tipos de *join point*, que são tratados de maneira uniforme. A função que implementa esta decisão pelo *Weaver* segue a semântica descrita no trabalho citado.

6.1.4 Transformações de código

O pré-processador realiza o processo de combinação por meio de uma série de transformações de código que enxertam as expressões definidas em declarações de *advice* nos *join points* adequados. As transformações são definidas principalmente de acordo com o tipo do *advice* (*before*, *after* ou *around*), contendo pequenas variações no tocante à maneira como foi estruturado o código monádico (com chamadas explícitas às funções $>>$ e $>>=$ ou por meio da notação *do*). Uma última variante para definição das transformações é referente às ligações de identificadores que expõem informações de contexto nos *join points*, apresentando implementações diversas para cada tipo de *join point*. As transformações implementadas pelo pré-processador aproveitam a natureza seqüencial do código monádico:

- *Advice do tipo before e after.* As transformações de código referentes a uma declaração de *advice* do tipo *before* ou *after* executam a expressão do *advice* antes e depois de seu *join point*, respectivamente. No caso de o *join point* ser definido por meio da notação *do* a expressão do *advice* será inserida como a primeira (*advice before*) ou a última (*advice after*) da expressão. No caso de o código conter chamadas explícitas a >> e >>=, utiliza-se uma chamada adicional à função >>, onde a expressão do *join point* será o operando da direita (*advice before*) ou da esquerda (*advice after*); o outro operando é a expressão do *advice*.

- *Advice do tipo around.* Declarações de *advice* do tipo *around* implicam em uma transformação de código que funciona da seguinte maneira: se a expressão do *advice* não contém uma chamada à função *proceed*, tal expressão simplesmente substitui a contida no *join point*; caso contrário, a expressão do *join point* será transformada numa expressão *let* (Seção 3.1.9) que contém uma função local (*proceed*) que contém o código do *join point* interceptado, e cuja expressão é a especificada na declaração de *advice*. Para esclarecer melhor, consideremos como fica o código da função *addUser*, vista na Seção 5.8.4, após a combinação apenas com a primeira versão do aspecto *ExceptionHandlingAspect* (Seção 5.8.6):

```
1. addUser :: String -> User -> IO String
2. addUser me entity@(User lg passwd rgts cpf nm)
3.  = let proceed me (User lg passwd rgts cpf nm)
4.        = withDB $ \db ->
5.          do insert db U.users (U.login    << constant lg #
6.                                U.password << constant passwd #
7.                                U.rights   << constant rgts #
8.                                U.cpf      << constant cpf #
9.                                U.name     << constant nm
10.                             )
11.            return "Ok"
12.    in catch (proceed me entity)
13.             (\e -> return ("Error:"++(show e)))
```

Note que a função *proceed* contém o código do *join point* interceptado, que pode ser chamada na expressão do *let*.

- *Ligações de identificadores.* As transformações que realizam as ligações de identificadores são bastantes distintas, no que diz respeito aos tipos de *join points*. No caso de execução de funções, existem diversas regras que são empregadas para casos específicos, juntamente com o uso do operador de padrões @ [1]. O código acima, da função *addUser* combinada apenas com a primeira versão do aspecto *ExceptionHandling*, representa um exemplo desta transformação. Na linha 2, vemos a

[1] Este operador estabelece uma relação de equivalência entre um identificador e um padrão. Por exemplo, o padrão *user @ (User lg passwd rgts cpf nm)* determina que seus operandos contenham o mesmo valor quando ocorrer um casamento de padrões.

utilização do operador @ para estabelecer a equivalência entre *entity* especificado no operador *args* e *User lg passwd rgts cpf nm*, parâmetro formal da função interceptada. O parâmetro *me* permanece inalterado uma vez que é especificado da mesma forma no operador *args* da declaração de *advice*.

Quanto a chamadas de funções, as ligações são realizadas por meio de uma expressão *let*. O *join point* é transformado numa expressão *let* que contém as ligações entre os padrões definidos no operador *args* e as expressões especificadas na chamada da função. Vejamos como fica o código da função *insertUser* (Seção 5.8.5) do módulo *Interface* na chamada da função *addUser* após a combinação apenas com a segunda versão do aspecto *ExceptionHandling* (Seção 5.8.7):

```
1. insertUser :: String -> IO ()
2. insertUser me =
3.   do (...)
4.
5.      res <- let proceed t1 t2 = addUser t1 t2
6.                 in let entity = User lg pw rgts nm cpf
7.                        in catch (proceed me entity)
8.                                 (\e -> return ("Error: " ++ (show e)))
9.      putStrLn res
```

Como o primeiro padrão especificado é idêntico ao primeiro argumento da chamada (*me*), não há necessidade de definir uma ligação entre eles; a única ligação ocorrida foi entre o padrão *entity* especificado no operador *args* e a expressão *User lg pw rgts nm cpf* (linha 6).

Uma observação acerca destas ligações é que a combinação não produz identificadores duplicados. Para isso, determinadas situações exigem a renomeação de identificadores dos padrões especificados em declarações de *advice*; a expressão definida lá, também é modificada para refletir a mudança ocorrida.

6.1.5 Verificação de tipos

Uma questão muito importante para o processo de combinação é a verificação de tipos. Isto inclui verificar os tipos do programa base em Haskell e dos aspectos separadamente e garantir que a combinação deles resulta em um programa com os tipos corretos.

O pré-processador de AspectH não suporta a verificação de tipos, entretanto, erros de tipos são detectados automaticamente pelo compilador Haskell original. O usuário utiliza o pré-processador para realizar a combinação de módulos Haskell e aspectos, e um compilador Haskell para compilar o programa resultante, monitorando os erros produzidos por ele. A verificação de tipos de aspectos faz parte dos trabalhos futuros (Seção 7.3), principalmente com o objetivo de facilitar a localização e correção de erros causados pelo uso incorreto de aspectos.

A verificação de tipos em AspectH representa uma tarefa desafiadora. Questões referentes a tipos incluem:

- verificar se os *join points* referenciados no código dos aspectos é monádico, isto é, se as funções selecionadas possuem tipo de retorno instância de *Monad*;
- verificar se a computação especificada em declarações de *advice* são compatíveis com os *join points* que eles modificam, isto é, se elas possuem os mesmos tipos;
- verificar se declarações de *advice* do tipo *after* ou *around* não modificam o tipo de retorno das funções que elas interceptam.

6.2 WEAVER: PROGRAMA BASE

A versão original do programa que realiza o processo de combinação (*Weaver*) foi codificado apenas em Haskell, como mencionado anteriormente. Reestruturamos este programa utilizando AspectH. O *Weaver* possui duas características que o tornam adequado para tal reestruturação, quais sejam, a de ser um programa monádico e de possuir deficiências quanto a modularidade de seus *crosscutting concerns*. Apresentamos nesta seção a funcionalidade básica do *Weaver*, já destituído do código referente aos *crosscutting concerns*.

O módulo *Weaver* contém a maior parte do código do pré-processador de AspectH. A maioria de suas funções é estruturada por meio da mônada definida no módulo *WeaverMonad*. As funções contidas no módulo *WeaverUtils* auxiliam no processo de combinação não possuindo, contudo, definições monádicas e são, portanto, livres da influência do código dos aspectos apresentados mais a frente neste capítulo.

Examinemos de início a definição da mônada do processo de combinação para que possamos entender melhor o funcionamento do módulo *Weaver* e seus *crosscutting concerns*.

6.2.1 Módulo WeaverMonad

As funcionalidades impuras pertinentes ao processo de combinação estão a cargo da mônada *WvM* do módulo *WeaverMonad*. Esta mônada é responsável por manter um ambiente com informações necessárias a combinação e por retornar todas as advertências (*warnings*) produzidas durante o processo. A mônada também responde pelo controle de exceções, evitando que o *Weaver* seja interrompido de forma abrupta em caso de erro. A combinação destas funcionalidades – estado e controle de exceções – foi realizada de forma manual (Seção 4.3), e não por meio de transformadores monádicos (Seção 4.4).

O ambiente adotado no *Weaver* é modelado pelo tipo algébrico *WvM*, apresentado a seguir:

```
1. data WvEnv = WvEnv {
2.         wvUnit           :: Maybe HsModularUnit,
3.         wvAspect         :: Maybe HsAspect,
4.         wvWithinNames    :: [(HsQName, [HsPat], [HsExpPcd])],
5.         wvWithinBindings :: [HsExpPcd],
6.         wvArgsBindings   :: [HsExpPcd],
7.         wvAspectAction   :: FiniteMap AspectId
```

```
8.                              [(Module, HsJoinPoint, HsAdvice)]
9. }
```

Durante a combinação, os *join points* de cada unidade modular (modulo ou aspecto) são inspecionados a fim de verificar se há código de aspectos que se aplique neles. O ambiente armazena tanto a unidade modular (*wvUnit*, linha 2) quanto o aspecto (*wvAspect*, linha 3) presentemente analisados. O campo *wvWithingNames* (linha 4) contém todos os *join points* dentro de uma declaração para facilitar a decisão gerada pelo operador *within* quanto a *join points* contidos dentro de uma determinada função; o campo *wvWithin-Bindings* (linha 5) contém as ligações provocadas pelo operador *within*; *wvArgsBindings* (linha 6) armazena as ligações geradas pelo operador *args*; e *wvAspectAction* mantém um mapeamento entre cada aspecto e os *join points* que ele modificou e que podem ser apresentados ao fim da combinação.

As advertências produzidas pelo *Weaver* ficam armazenadas em um objeto do tipo *WvWarnings*:

```
type WvWarnings = [String]
```

que é uma lista de *Strings*, com a descrição de cada advertência.

O tipo de dado *WvState* é empregado para manipular exceções e é semelhante a *Maybe* (Seção 4.1.4), com a diferença de que o estado de erro contém uma descrição da exceção ocorrida:

```
data WvState a = WvSucess a | WvError String
```

A mônada empregada no processo de combinação é representada pelo construtor *WvM*, visto a seguir:

```
newtype WvM result
  = WvM (WvEnv -> WvWarnings -> WvState (result, WvEnv, WvWarnings))
```

O tipo *WvM* é uma função que recebe como argumento o ambiente e as advertências e retorna um valor do tipo *WvState*. Em caso de sucesso esta função retorna seu resultado (*result*), juntamente com o ambiente e as advertências geradas.

Em determinadas situações é interessante extrair o valor contido em uma mônada. No caso de *WvM* este valor é uma função que pode, por exemplo, ser aplicada no início do processo a um ambiente inicial e a uma lista (vazia) de advertências. A função *unWvM* a seguir possui este propósito:

```
unWvM :: WvM a -> WvEnv -> WvWarnings ->
                     WvState (result, WvEnv, WvWarnings)
unWvM (WvM f) = f
```

Para que *WvM* seja considerado uma mônada é necessário uma declaração que o defina como instância de *Monad*, contendo implementações para as funções *return* e >>=:

```
1. instance Monad WvM where
2.  return :: a -> WvM a
3.  return result
4.     = WvM (\env warns -> WvSucess (result, env, warns))
5.
6.  (>>=) :: WvM a -> (a -> WvM b) -> WvM b
7.  (WvM m1) >>= m2
8.     = WvM (\ env warns ->
9.               case m1 env warns of
10.                WvSucess (result, env', warns')
11.                   -> unWvM (m2 result) env' warns'
12.                WvError s -> WvError s
13.          )
```

A função *return* (linhas 2 a 4) apenas injeta um valor na mônada *WvM*. >>= (linha 6 em diante) funciona da seguinte maneira: caso a computação *WvM m1* ocorra com sucesso (*WvSucess*, linhas 10 e 11) seu resultado é passado para a computação seguinte *m2*; se *WvM m1* produzir um erro (*WvError*, linha 12), este estado será propagado. Observe que a chamada à *unWvM* (linha 11) obtém a função contida na mônada que resulta da aplicação de *m2* a *result* e que é, então, aplicada ao ambiente e às advertências produzidas pela computação *WvM m1 – env'* e *warns'*, respectivamente.

Para interromper o processo de combinação em caso de erro basta fazer uma chamada à função *wvRaise* passando como argumento a descrição do erro ocorrido:

```
wvRaise :: String -> WvM a
wvRaise s = WvM (\ env warns -> WvError s)
```

A função *wvWarn* a seguir está a cargo da emissão de advertências:

```
wvWarn :: WvWarning -> WvM ()
wvWarn warn
  = WvM (\env warns -> WvSucess ((), env, warn : warns))
```

Note que a advertência (*warn*) passada como argumento para *wvWarn* é posta na cabeça da lista de advertências produzida até então pela combinação (*warn : warns*).

Para manipular o ambiente, o módulo *WvM* define as funções *getEnv* e *updEnv*: *getEnv* obtém o estado atual do ambiente para ser examinado e *updEnv* tem como argumento uma função que é aplicada ao ambiente atual no afã de atualizá-lo:

```
getEnv :: WvM WvEnv
getEnv = WvM (\ env warn -> WvSucess (env, env, warn)

updEnv :: (WvEnv -> WvEnv) -> WvM ()
updEnv upd = WvM (\ env warn -> WvSucess ((), upd env, warn))
```

O módulo *WeaverMonad* contém outras funções que facilitam a utilização da mônada *WvM* e que serão omitidas por não influenciarem diretamente no entendimento geral da discussão.

6.2.2 Módulo Weaver

O funcionamento do código do módulo *Weaver* que utiliza a mônada *WvM* gira em torno do ambiente mantido por ela. As funções acessam e atualizam o ambiente como parte de sua execução. Além disso, as advertências e os erros percebidos pelo *Weaver* devem também estar presentes, juntamente com a funcionalidade principal relacionada a cada função.

As atualizações realizadas no ambiente representam parte considerável do código destas funções e determinadas atualizações repetem-se por diversas funções. O código torna-se confuso em algumas situações e a modificação de código, particularmente no responsável pelas atualizações no ambiente é bastante inconveniente. Este é, portanto, o primeiro *crosscutting concern* identificado no código do *Weaver* e que deve ter sua implementação realizada em aspectos. Os outros se referem ao controle de exceções e a emissão de advertências durante o processo de combinação.

O código do *Weaver* é consideravelmente extenso, portanto, apresentamos apenas uma amostra relevante de fragmentos de suas funções e onde nelas se aplica o código necessário para realizar seus *crosscutting concerns*. Sua implementação propriamente dita é feita de forma separada em aspectos, discutidos na Seção 6.3.

Como parte fundamental do processo temos uma função que combina os aspectos com as unidades modulares (módulos ou aspectos). Esta função, *weaveModularUnit*, recebe um aspecto e uma unidade modular e retorna a unidade modular, possivelmente modificada. Apresentamos a seguir apenas uma de suas equações cuja unidade modular é referente a módulos convencionais (a outra, referente a aspectos, é similar):

```
1. weaveModularUnit :: HsAspect -> HsModularUnit -> WvM HsModularUnit
2. weaveModularUnit
3.   apt@(HsAspect src aptId _ _ _)
4.   mod@(HsHaskellModule (HsModule src' mdlId exts imts decls))
5.  = let applyAdvice :: HsDecl -> [HsAdvice] -> WvM HsDecl
6.        applyAdvice dec ((tyApt, pcd, advice):as) =
7.           do weavedDecl <- weaveDecl (tyApt, pcd, advice) dec
8.              applyAdvice weavedDecl as
9.        applyAdvice dec [] = return dec
          (...)

10.   in do let advs = getPiecesOfAdvice apt
11.         weavedDecls <- mapWv (\dec -> applyAdvice dec advs) decls
            (...)

12.         return $
13.          HsHaskellModule (HsModule src' mdlId exts imts' weavedDecls)
```

Esta função "aplica" todas as declarações de *advice* (*advs*, linha 10) do aspecto (apt, linha 3) às declarações do módulo (*decls*, linha 4). A função local *applyAdvice* (linhas 5 a 9) chama a função *weaveDecl* que combina as declarações de *advice* com as declarações

do módulo. Observe que a função *applyAdvice* é aplicada às declarações do módulo e o resultado é ligado a *weavedDecls* (linha 11) [2]. Ao final (linhas 12 e 13), o módulo é retornado com suas declarações possivelmente modificadas.

A única atualização a ser feita no ambiente, antes da execução da função *weaveModularUnit*, é em seus campos *wvUnit* e *wvAspect* , que passam a conter o aspecto (*apt*, linha 3) e o módulo (*mod*, linha 4), presentemente sendo combinados.

Uma declaração é modelada pelo tipo algébrico *HsDecl* que contém diversos construtores, correspondentes às declarações de Haskell e de AspectH: declarações de tipos, de classe, de *advice*, etc. A declaração que nos interessa aqui é a que representa funções, correspondente ao construtor *HsFunBind*. Como vimos, uma função pode ter diversas equações. Cada equação é modelada pelo tipo algébrico *HsMatch*. A seguir temos o construtor *HsFunBind* de *HsDecl* correspondente às equações de uma função e o tipo *HsMatch*:

```
data HsDecl = (...)
            | HsFunBind [HsMatch]

data HsMatch
 = HsMatch SrcLoc HsName [HsPat] HsRhs [HsDecl]
```

Note que o construtor *HsFunBind* possui uma lista com cada equação (*match*) que define uma função. Uma equação possui um identificador (*HsName*), parâmetros formais ([*HsPat*]), a expressão "do lado direito" (*HsRhs*) e as declarações locais feitas na cláusula *where* ([*HsDecl*]).

A função *weaveDecl* lida diretamente com o tipo de dado *HsDecl* e possui equações referentes a declarações de função e de *advice*, e uma terceira que lida com o caso geral, que é não realizar transformação alguma. Ela primeiramente verifica se a execução de uma função é selecionada pelo *pointcut* de uma declaração de *advice* e, em caso afirmativo, a expressão do *advice* é combinada com a equação. Vejamos a seguir a equação de *weaveDecl* correspondente ao construtor *HsFunBind*:

```
1. weaveDecl :: HsAdvice -> HsDecl -> WvM HsDecl
2. weaveDecl advice@(tyAdv, pcd, advExp) (HsFunBind mts)
3.    = let applyToMatch :: Module -> HsMatch -> WvM HsMatch
4.          applyToMatch
5.           modId
6.           mtc@(HsMatch s nm jpDecs (HsUnGuardedRhs jpRhsExp) decls)
7.             = do let jp = ExecutionJP (Qual modId nm, jpDecs)
8.                  result <- matchPcd jp pcd
9.                  match <- weaveMatch result modId advice mtc
10.                 return match
11.      in do (...)

12.           weavedMatches <- mapWv (applyToMatch modId) mts
13.           return $ HsFunBind weavedMatches
```

[2] A função *mapWv* do módulo *WeaverMonad* aplica uma função aos elementos de uma lista.

weaveDecl possui uma função local, *applyToMatch* (linhas 3 a 10), que segue o processo de combinação em todas as equações (*mts*, linha 2) de uma função. As equações modificadas são ligadas ao identificador *weavedMatches* (linha 12). Observe na linha 8 a chamada à função *matchPcd* que decide se o *join point* em questão (*jp*, linha 7) foi selecionado; seu resultado é ligado ao identificador *result* que é então passado para a função *weaveMatch* (linha 9) que realiza as modificações adequadas na equação (*mtc*, linha 6).

Existem duas atualizações a serem feitas no ambiente aqui: antes da execução de *applyToMatch* a lista *wvWithinNames* do ambiente deve receber a equação sendo modificada na cabeça da lista; ao final, esta declaração deve ser removida.

A função *weaveMatch* é responsável por modificar a equação quando ela é selecionada pela declaração de *advice*. Antes disso, entretanto, ela verifica também se os *join points* contidos na expressão do lado direito desta equação e de suas declarações locais definidas pela expressão *where* são selecionadas pela declaração de *advice*. Isto é independente de a equação ser selecionada ou não. Vejamos a seguir a função *weaveMatch*:

```
1. weaveMatch :: Bool -> Module -> HsAdvice -> HsMatch -> WvM HsMatch
2. weaveMatch result modId advice@(tyAdv, pcd, advExp)
3.            (HsMatch s nm jpDecs (HsUnGuardedRhs jpRhsExp) decls)
4.   = let (...)

5.     in do jpRhsExp' <- weaveExp s modId advice jpRhsExp
6.           decls' <- mapWv (weaveDecl advice) decls
             (...)

7.           if result
8.            then
9.             do (...)

10.              (jpDecs'', advExp'')
11.                <- applyBindings advExp' jpDecs' argsBnds
12.              let jpRhsExp''
13.                = weaveFunBind s jpDecs'' jpRhsExp'
14.                               tyAdv advExp''
15.              return $
16.               HsMatch s nm jpDecs'' (HsUnGuardedRhs jpRhsExp'')
17.                       decls'
18.          else
19.           return $
20.            HsMatch s nm jpDecs' (HsUnGuardedRhs jpRhsExp') decls'
```

Na linha 5 os *join points* da expressão do lado direito da equação (*jpRhsExp*) são vasculhados para verificar se o *advice* em questão (*advice*, linha 2) se aplica a eles; note a chamada à função *weaveExp*, cujo resultado é ligado a *jpRhsExp'*: esta função realiza recursão na estrutura das expressões Haskell em busca de chamadas e execução de funções. Na linha 6, as declarações locais da equação (*decls*, linha 3) são aplicadas à

função *weaveDecl* e seu resultado ligado a *decls'*. O condicional (linha 7 em diante) verifica se a equação foi interceptada (valor contido no parâmetro formal *result*). Em caso afirmativo, os parâmetros formais desta equação são combinados com os identificadores especificados no *pointcut* (*argsBnds*) por meio da função local *applyBindings* (linhas 10 e 11) e a expressão do lado direito da equação é combinada com a declaração de *advice* por meio da função *weaveFunBind* (linhas 12 a 14).

As atualizações na função *weaveMatch* dizem respeito ao mapeamento *wvAspectAction* do ambiente caso alguma declaração de *advice* modifique a unidade modular. Esta atualização é a mesma que ocorre na função *weaveExp*.

A função *matchPcd* é responsável por decidir se um *join point* é selecionado por um *pointcut*. Ela realiza uma recursão estrutural no *pointcut*, possuindo diversas equações que tratam dos dois tipos de *join point* e dos operadores de *pointcut*.

Como amostra vejamos três equações da função *matchPcd*:

```
1.  matchPcd :: HsJoinPoint -> HsPcd -> WvM Bool
2.  matchPcd (ExecutionJP (jpName, _)) (HsPcdDef (HsOpExecution qName))
3.   = do let matched = jpName == qName
4.         return matched
5.  matchPcd (ExecutionJP (_, _)) (HsPcdDef (HsOpWithin qname pats))
6.   = let matchWn :: [(HsQName, [HsPat], [HsExpPcd])] -> HsQName
7.                    -> Bool
8.         matchWn [] _ = False
9.         matchWn ((qn, pts, _):wns) qname
10.         = ( qname == qn &&
11.             ((length pts) == (length pats)) &&
12.             ((and . (map (\(pat1, pat2) -> matchPattern pat1 pat2)))
13.                 $  (zip pts pats)) )
14.             || matchWn wns qname
15.    in do WvEnv _ _ wNames _ _ _  <- getEnv
16.          let matched = matchWn wNames qname
17.          return matched
18. matchPcd jp (HsAndPcd pcd1 pcd2)
19.  = do matched1 <- matchPcd jp pcd1
20.       matched2 <- matchPcd jp pcd2
21.       let matched = matched1 && matched2
22.       return matched
```

A primeira equação (linhas 2 a 4) verifica se a execução de uma função é selecionada pelo operador de *pointcuts execution*. Para tanto, a função deve possuir o mesmo identificador (*jpName*, linha 2) especificado no operador *execution* (*qName*, linha 2). Nesta equação é necessário que o *join point* avaliado seja referente a uma função exportada pelo módulo. Caso não seja, uma exceção deve ser disparada e o processo de combinação interrompido.

A segunda equação (linhas 5 a 17) averigua se a execução de uma função é selecionada pelo operador *within*. A função local *matchWn* (linhas 6 a 14) realiza os testes necessários, verificando todas as equações presentes na lista *wvWithinNames*. Note na

linha 15 a chamada a *getEnv*: ela obtém o valor do campo *wvWithinNames* (ligado a *wNames*) que é, assim, passado para a função *matchWn* (linha 16). Caso alguma função em *wvWithinNames* seja selecionada, o campo *wvWithinBindings* deve ser modificado para adicionar as possíveis ligações produzidas por este operador.

A terceira equação (linhas 18 a 22) é referente ao operador binário de *pointcuts* &&. Aqui apenas é feita a recursão que verifica se o *join point* é selecionado por seus operandos (linhas 19 e 20) e então é feita a conjunção dos resultados (linha 21).

6.3 ASPECTOS

Os *crosscutting concerns* de AspectH, descritos na Seção 6.2, estão implementados em três aspectos. O primeiro deles – e mais extenso – *StateAspect*, contém o código responsável pelas atualizações realizadas no ambiente mantido pela mônada *WvM*; *ExceptionAspect* compreende o código referente ao controle de exceções do sistema; por fim, *WarningAspect* abrange o código que emite as advertências geradas pelo *Weaver*.

Como fizemos com o código base, apresentamos aqui apenas uma amostra do código destes aspectos, que influenciam as funções do código base descritas na Seção 6.2.2. As próximas três subseções estão dedicadas aos aspectos *StateAspect*, *ExceptionAspect*, e *WarningAspect*, respectivamente.

6.3.1 Aspecto StateAspect

O aspecto *StateAspect* modifica as funções do módulo *Weaver*, com o intuito de realizar as atualizações necessárias no ambiente do *Weaver*. As declarações de *advice* deste aspecto provêem uma implementação modular para este *crosscutting concern* do sistema.

Na Seção 6.2.2 começamos a descrição do código base pela função *weaveModularUnit*. A declaração de *advice* a seguir realiza a atualização necessária no ambiente, referente aos campos *wvUnit* e *wvAspect*:

```
1. before: execution Weaver.weaveModularUnit &&
2.         args apt unit
3.  = let upd :: WvEnv -> WvEnv
4.        upd env = env {wvUnit=Just unit, wvAspect=Just apt}
5.    in updEnv upd
```

A expressão do *advice before* chama a função *updEnv* (linha 5) que aplica a função local *upd* ao ambiente, realizando a atualização necessária. Esta declaração intercepta as duas equações da função *weaveModularUnit*.

O *advice around* a seguir aplica as funções locais *init* (linhas 4 a 8) e *end* (linhas 9 a 12) ao ambiente, antes e depois da execução da função *applyToMatch*, local a *weaveDecl*:

```
1. around: execution Weaver.applyToMatch           &&
2.         within Weaver.weaveDecl _ _             &&
3.         args modId mtc@(HsMatch s nm jpDecs _ _)
4.   = let init :: WvEnv -> WvEnv
```

```
5.          init env = let WvEnv _ _ wvWNs wvBds _ _ = env
6.                         wvWNs' = (Qual modId nm, jpDecs, []):wvWNs
7.                         wvBds' = []
8.                     in env {wvWithinNames=wvWNs'}
9.          end :: WvEnv -> WvEnv
10.         end env = let WvEnv _ _ wvWNs _ _ _ = env
11.                       wvWNs' = tail wvWNs
12.                   in env {wvWithinNames=wvWNs'}
13.     in do updEnv init
14.           res <- proceed modId mtc
15.           updEnv end
16.           return res
```

Os operadores de *pointcut execution*, *within* e *args* são compostos por meio de && (linhas 1 a 3) para selecionar a função *applyToMatch* local a *weaveDecl*. Note as chamadas a *updEnv* que aplicam a função *init* (linha 13) e *end* (linha 15) ao ambiente antes e depois da chamada a *proceed* (linha 14) que reinstala a computação interceptada. *init* adiciona a equação em questão na cabeça da lista *wvWithinNames* e *end* a remove. Ao final o resultado produzido por *proceed* (ligado ao identificador *res*) é retornado (linha 16).

Quando um aspecto modifica um determinado *join point*, o mapeamento *wvAspectAction* do ambiente deve ser modificado para registrar este fato. O *advice before* a seguir realiza esta atualização:

```
1. before: execution Weaver.weaveMatch &&
2.         args result modId advice (HsMatch _ nm jpDecs _ _)
3.    = let upd :: Bool -> WvEnv -> WvEnv
4.          upd True env
5.            = let jp = ExecutionJP (Qual modId nm, jpDecs)
6.                  WvEnv _ (Just apt) _ _ _ fm = env
7.                  HsAspect _ aptId _ _ _ = apt
8.                  key = aptId
9.                  value = case lookupFM fm key of
10.                           Just value -> value
11.                           Nothing -> []
12.                 value' = (modId, jp, advice):value
13.                 fm' = addToFM fm key value'
14.              in env {wvAdvice=fm'}
15.         upd False env = env
16.     in updEnv (upd result)
```

Ele aplica a função local *upd* ao ambiente passando o valor contido no parâmetro *result*, que decide se o *join point* foi modificado ou não pelo aspecto. Observe que *upd* possui duas equações (linhas 4 a 14 e linha 15): a primeira inclui no mapeamento as informações referentes ao aspecto e aos *join points*; a segunda apenas retorna o ambiente sem modificações, já que o *join point* não foi interceptado.

O *advice around* a seguir intercepta a execução de quatro equações da função *matchPcd*, referentes ao operador de *pointcut within* (com e sem coringa):

```
1. around: execution Weaver.matchPcd &&
2.         ( args
3.            jp pcd@(HsPcdDef pcdExp@(HsOpWithin _ _)) ||
4.           args
5.            jp pcd@(HsPcdDef pcdExp@(HsOpWithinWildcard _ _)) )
6.  = let upd :: Bool -> WvEnv -> WvEnv
7.        upd True env
8.          = let WvEnv _ _ _ wns _ _ = env
9.                wns' = (pcdExp):wns
10.            in env { withinBindings= wns' }
11.       upd False env = env
12.    in do res <- proceed jp pcd
13.          updEnv (upd res)
14.          return res
```

O *pointcut* emprega o operador *execution* para interceptar a função *matchPcd*. Dois operadores *args* (linhas 2 a 5) são compostos por meio do operador || para selecionar as quatro equações de interesse. Os *join points* são ligados ao mesmo identificador *jp*, enquanto que *pcd* representada o operador *within* (com e sem coringa); estes identificadores são utilizados como argumentos na chamada a *proceed* (linha 12) que retoma a execução das funções interceptadas. A função local *upd* (linhas 6 a 11) é aplicada ao ambiente para atualizar a lista *withinBindings* no caso da execução de alguma função ter sido selecionada pelo operador *within*.

6.3.2 Aspecto ExceptionAspect

O aspecto *ExceptionAspect* é responsável pelo controle de exceções do sistema. Ele modifica o código das funções que podem eventualmente causar erros, removendo o tratamento feito a tais erros do código base.

Apresentamos a seguir uma declaração de *advice* do tipo *around* que seleciona duas equações da função *matchPcd* referentes ao operador de *pointcuts execution* (com e sem coringa). As funções selecionadas por ele devem ser exportadas pelo módulo; caso não sejam o processo de combinação deve ser interrompido:

```
1. around: execution Weaver.matchPcd &&
2.         ( args jp@(ExecutionJP (jpName, _))
3.                pcd@(HsPcdDef (HsOpExecution _)) ||
4.           args jp@(ExecutionJP (jpName, _))
5.                pcd@(HsPcdDef (HsOpExecutionWildcard _)) )
6.  = do WvEnv (Just unit) _ _ _ _ _ <- getEnv
7.       res <- proceed jp pcd
8.       if res && (not $ isExported unit jpName)
9.          then
```

```
10.            wvRaise $ "Function " ++ prettyPrint jpName ++
11.                      " is not exported!"
12.          else return res
```

Na linha 6, por meio de *getEnv*, obtém-se a unidade modular (*unit*) presentemente modificada, contida no ambiente. As funções interceptadas são reativadas através de *proceed* (linha 7), passando-se o *join point* (*jp*) e o *pointcut* (*pcd*) como argumentos. O condicional (linha 8 em diante) verifica se a função em questão (*jpName*) é exportada por meio de *isExported* (linha 8) que tem como argumento a unidade modular (*unit*) e a função interceptada: caso tal função não seja exportada a função *wvRaise* (linhas 10 e 11) é chamada, interrompendo o processo com uma mensagem de erro.

6.3.3 Aspecto WarningAspect

A emissão de advertências pelo *Weaver* está a cargo do aspecto *WarningAspect*. Ele provê uma implementação modular para este *crosscutting concern*, que de outra forma estorvaria o código base do sistema.

A função *weaveArgsInFunBind* do módulo *Weaver* combina os parâmetros formais de uma função com os identificadores especificados em *pointcuts*, para que a informação de interesse seja exposta adequadamente. Como mencionado (Seção 6.1.4), este processo pode requerer a renomeação de algum identificador para resolver conflitos de nome. Neste caso a expressão que define o *advice* deve ser modificada para refletir esta alteração. Vejamos a seguir o *advice around* que seleciona a execução da função *weaveArgsInFunBind*:

```
1. around: execution Weaver.weaveArgsInFunBind &&
2.         args advice jpDecs pcdArgs
3.  = let msg = "The expression defining the advice " ++
4.              (prettyPrint advice) ++
5.              " was redefined to resolve name clashes"
6.     in do (jpDecls', advice') <- proceed advice jpDecs pcdArgs
7.           if advice /= advice'
8.            then wvWarn msg
9.            else return ()
10.          return (jpDecls', advice')
```

Na linha 6 é feita a chamada à *proceed* ligando seu resultado a uma dupla (que contém a expressão do *advice* como segundo componente, possivelmente alterada). O condicional (linhas 7 a 9) averigua se a expressão do *advice* foi modificada, emitindo a mensagem (*msg*, linha 8) com a descrição da advertência produzida, caso esta expressão tenha sido modificada.

6.4 DISCUSSÃO

A implementação do *Weaver* utilizando aspectos apresenta diversas vantagens em relação a versão original. A mais expressiva delas é, certamente, a legibilidade do código

base. Suas funções foram bastante simplificadas, uma vez que desempenham apenas seu propósito básico. Além disso, é interessante ressaltar que o funcionamento do *Weaver*, se comparado com a versão original apenas em Haskell, manteve-se inalterado. O programa apresentado na Seção 5.8 foi combinado utilizando-se as duas versões do *Weaver*, apresentando como resultado programas exatamente idênticos.

O emprego de aspectos também favoreceu a manutenção do programa, tendo em vista que algumas ações idênticas puderam ser incorporadas em uma única declaração de *advice*. Entretanto, esta melhoria não é considerada crítica, já que existem determinadas ações que, mesmo idênticas e que se repetiam em mais de uma função, não puderam ser expressas em um mesmo *advice*.

Outro ponto favorável é o fato de os aspectos do sistema serem unidades perfeitamente separáveis, podendo ser removidos para gerar automaticamente versões diferentes do programa que facilitam tarefas como por exemplo a realização de testes, onde o programa pode ser verificado sem a presença de alguns aspectos.

Os aspectos encontrados no *Weaver* de AspectH são típicos de programas do mesmo gênero, como pré-processadores e compiladores. Um exemplo é o próprio compilador de Haskell GHC [Tea04a]. Este compilador possui as mesmas características negativas encontradas no estudo de caso deste capítulo, principalmente no tocante à complexidade resultante de um código pouco modular, onde os *crosscutting concerns* estão espalhados e misturados no código base. Uma parte deste compilador, responsável pela eliminação das conveniências sintáticas (o *deSugar*) é estruturada por meio de uma mônada bastante similar à *WvM*. Necessidades tais como atualização do ambiente mantido pela mônada, controle de exceções e emissão de advertências (entre outras), também são encontradas e são bastante próximas às discutidas aqui. Este capítulo pode servir, então, para nortear a reestruturação de programas como o compilador GHC empregando-se AspectH, com o objetivo de gerar programas mais legíveis, mais fáceis de manter, reusar, etc.

AspectH demonstrou eficiência nos dois programas apresentados neste texto quanto à definição de programas mais modulares, onde seus *crosscutting concerns* puderam ser separados do código base, de modo a obter todos os resultados positivos que advêm daí. Entretanto, é interessante considerar uma característica negativa pertinente à linguagem, oriunda da própria abordagem e também encontrada em outras linguagens como AspectJ, que é o fato de o código dos aspectos estar por demais atrelado à representação textual do código base. Isto quer dizer, por exemplo, que a simples modificação do nome de uma função pode comprometer o funcionamento de um aspecto que atua nesta função. Em algumas situações, pode-se tornar complicada a sincronização entre o programa base e os aspectos. Além disso, tal limitação apresenta empecilhos na definição de declarações de *advice*. Em alguns casos, como já mencionado anteriormente, é necessária a especificação de mais de uma declaração de *advice* com a mesma computação, uma vez que os *join points* afetados não são compatíveis, o que significa, na maioria dos casos, que possuem assinaturas diferentes. Indo além, seria interessante a implementação de um processo, por assim dizer, menos sintático onde fosse possível a especificação mais genérica do código dos aspectos.

Esta abordagem dificulta o reuso do código dos aspectos. Idealmente eles deveriam ser utilizados em diversos "programas base", onde sua execução estaria condicionada a

informações menos ligadas a como um programa em particular foi escrito.

Outro fator que prejudicou uma avaliação mais ampla da linguagem foi o fato de as questões de tipo terem sido negligenciadas. A verificação de tipos, além de tornar mais cômodo o processo, facilitando a detecção de erros provocados pelo uso incorreto de aspectos, pode revelar e esclarecer diversas sutilezas mantidas inexploradas neste trabalho, e que permitiriam um entendimento mais amplo do processo.

É interessante também analisar a importante questão referente a prova de propriedades em programas Haskell modificados por AspectH. AspectH pode apresentar dificuldades em provar que determinadas propriedades são satisfeitas em um programa, devido à falta de localidade do código definido em aspectos. Indo mais além, é possível afirmar que os aspectos tornam a prova de propriedades em programas totalmente inviável, uma vez que sua presença destrói completamente a transparência referencial de um programa. É alto o preço exigido pelos aspecto: o maior grau de modularidade provido é compensado por uma maior dificuldade de entendimento que torna impraticável análises rigorosas referentes a propriedades dos programas.

A despeito das limitações e deficiências da abordagem, AspectH atende à expectativa inicial da pesquisa que é, principalmente, a de ser um instrumento de avaliação dos benefícios que uma abordagem AOP, bastante semelhante a implementada em outros paradigmas, pode trazer para uma linguagem funcional. AspectH é um ferramenta de transformação de código monádico que representa uma alternativa viável a outras linguagens para atualização de programas funcionais (Seção 7.2.5) apresentando diversas vantagens, podendo ser empregada em contextos e programas maiores.

6.5 INTRODUÇÃO SELETIVA E TARDIA DE MÔNADAS

O desenvolvimento de um programa funcional que envolve funcionalidades impuras – e que, portanto, necessita de código monádico – pode ser bastante facilitado se a introdução das definições monádicas for evitada em sua fase inicial. Isto porque escrever código monádico não é tão conveniente quanto escrever código funcional não-monádico. Em seu trabalho sobre monadificação de programas funcionais [ER04], Erwig e Ren fazem a seguinte afirmação acerca da programação monádica: "mônadas não são apenas difíceis para iniciantes, elas são inconvenientes para programadores experientes também – por exemplo, elas forçam o programador a especificar uma ordem de avaliação".

Assim, o desenvolvimento do programa deve iniciar com a implementação de funções não-monádicas e, posteriormente, o programa é transformado para uma forma monádica. A introdução das definições monádicas é feita de forma seletiva, isto é, apenas um subconjunto das funções do programa será modificado para a inclusão de código monádico. O programador deve escolher de forma adequada as funções que necessitam realmente estar em forma monádica. Após a escolha das funções de interesse, é preciso identificar as outras funções no programa que contenham chamadas a estas funções, para que sejam também monadificadas. Isto pode, no entanto, causar um problema de proliferação de funções monádicas. Para resolvê-lo basta apenas extrair o valor contido no tipo monádico, evitando que funções que contenham chamadas a funções monádicas (ou chamadas recursivas), e que possivelmente não precisam de fato estarem em forma monádica, sejam monadificadas no processo.

O objetivo da monadificação (processo que introduz definições monádicas em um programa funcional estrito, isto é, que não possui definições monádicas) é transformar uma dada função f de tipo $t_1 \rightarrow t_2 \rightarrow ... \rightarrow t_k \rightarrow t$ em uma função f' de tipo $t_1 \rightarrow t_2 \rightarrow ... \rightarrow t_k \rightarrow m\ t$, onde m é um construtor de tipo monádico [ER04]. A expressão que define a função deve ser também modificada. Em alguns casos esta modificação se resume a substituição do valor retornado pela função por uma chamada a função *return* passando o valor substituído como argumento, isto é, deve-se injetar este valor na mônada. Em outros, pode haver a necessidade de transformar toda a expressão empregando-se a notação *do* (ou chamadas a >>= e >>), devido a existência de chamadas de função dentro da expressão que possuem tipos monádicos, que incluem as chamadas recursivas.

O pré-processador de AspectH representa um exemplo de emprego da estratégia de introdução tardia e seletiva de mônadas. A princípio nenhuma função do módulo *Weaver* estava em forma monádica. Parte considerável de sua implementação foi realizada antes mesmo de a mônada *WvM* ter sido desenvolvida. Isto favoreceu bastante o desenvolvimento do sistema, já que foi possível trabalhar com as funções em sua forma estrita por mais tempo até a definição de versões mais estáveis das mesmas. Ademais, as preocupações referentes às funcionalidades impuras foram proteladas ao máximo, permitindo que o programador se concentrasse exclusivamente na funcionalidade básica do *Weaver* por um maior período, o que resultou, conseqüentemente, em um maior rendimento nesta fase.

A própria definição da mônada foi facilitada com esta estratégia, uma vez que as idéias acerca do que realmente era necessário para sua implementação estavam já suficientemente maduras quando deu-se início seu desenvolvimento. É provável que, caso sua definição tivesse sido providenciada desde o início, quando carecia de solidez o próprio entendimento do processo como um todo, a mônada passaria por um número de modificações maior do que o observado durante a implementação do *Weaver*. De fato, foram poucas as mudanças realizadas na definição da mônada *WvM*, nenhuma das quais considerada crítica a ponto de mudar os rumos da implementação.

Um outro ponto interessante a discutir é a utilização de um ambiente de programação que fornecesse um operador para monadificação automática de programas funcionais. O programador indicaria a mônada de interesse e o conjunto de funções que deseja monadificar. Tal ambiente proporcionaria diversas vantagens, uma das quais é versatilidade, já que se poderia utilizar mônadas diferentes, suportando funcionalidades distintas, criando sistemas com propósitos variados de forma automática.

Tendo em vista o papel fundamental de ambientes de programação para o suporte de linguagens orientadas a aspectos, o ambiente supra-citado poderia incluir, além deste operador para monadificação, suporte amplo a AspectH, que permitisse observar graficamente a influência de aspectos no código base, identificando os *join points* afetados por mais de uma declaração de advice e que facilitasse a detecção e correção de erros provocados pelo uso incorreto de aspectos, por exemplo. Este ambiente tornaria todo o processo, que inclui o desenvolvimento inicial sem mônadas, com a posterior monadificação seletiva (e automática) do programa e a implementação dos *crosscutting concerns* em aspectos, bem mais atrativo, eficiente e seguro. A implementação de tal ambiente é um dos principais trabalhos futuros (Seção 7.3) desta pesquisa.

CAPÍTULO 7

CONCLUSÕES E TRABALHOS FUTUROS

Nesta dissertação apresentamos AspectH, uma extensão orientada a aspectos da linguagem funcional Haskell, que permite a descrição modular e a composição de *crosscutting concerns* de programas monádicos. AspectH implementa AOP através de *pointcuts* e *advice* que modificam o código de módulos do programa base. Tal modificação se dá pela execução de computações adicionais que ocorrem em resposta a eventos durante a execução do programa, que em AspectH são chamadas e execuções de função. Uma declaração de *advice* é responsável pela especificação tanto da computação que se deseja instalar quanto dos locais no programa (os *join points* de interesse) onde esta computação executa, por meio de seu *pointcut*. Os aspectos contêm as declarações de *advice* que em conjunto capturam a implementação de *crosscutting concerns* em um programa.

Como vimos, AspectH é uma ferramenta puramente estática, já que todas as decisões referentes a execução do código dos aspectos são feitas estaticamente, possuindo, entretanto, suporte à quantificação estática e dinâmica. Além disso, seu *Weaver* modifica código apenas de programas estruturados por meio de mônadas. A própria natureza seqüencial da noção de mônada a torna um ambiente favorável a especificação de computações adicionais que definem uma ordem relativa de execução. Desta forma, a AspectH é uma linguagem adequada para adição de ações monádicas, que são computações que se adiciona em um código monádico com o objetivo de atualizá-lo.

No Capítulo 5, descrevemos as características gerais da linguagem. Por meio de um pequeno programa expusemos como é feita a implementação de exemplos clássicos de *crosscutting concerns* em aspectos resultando em um programa final bem mais legível devido ao maior grau de modularidade provido pelos aspectos. O programa também prestou-se a uma apreciação das distinções existentes entre os dois tipos de *join point* permitidos em AspectH, onde pudemos observar em um exemplo concreto os operadores de *pointcut* existentes na linguagem.

Apresentamos no Capítulo 6, como estudo de caso para AspectH, um programa com dimensões maiores do que o exposto na Seção 5.8 e com exemplos de aspectos diferentes dos vistos lá. Este programa foi o próprio *Weaver* da linguagem que se mostrou um exemplo interessante por possuir necessidades freqüentemente encontradas em outros programas do gênero, como, por exemplo, compiladores. O programa original foi reestruturado de modo a criar um sistema mais modular, separando os *crosscutting concerns* do programa base, por meio de sua definição em aspectos. Assim discutimos os resultados obtidos neste estudo de caso especificamente e a linguagem AspectH de modo geral. O capítulo ainda descreve a estratégia adotada na implementação do *Weaver* onde se evitou a utilização de construções monádicas desde o início. A introdução é realizada apenas quando se tem versões mais consistentes das funções do programa possibilitando assim um maior rendimento, já que os inconvenientes envolvidos com as construções monádicas são evitados nesta fase inicial. Além disso, esta introdução é feita de forma

seletiva somente envolvendo um subconjunto destas funções. Após a monadificação das funções utiliza-se AspectH para modificar o programa, separando em aspectos todo o código que de outra forma estaria misturado com a funcionalidade principal do código base.

O restante deste capítulo está assim organizado: na próxima seção apresentamos as principais contribuições deste trabalho; a seguir (Seção 7.2) relatamos os trabalhos que, de alguma forma, exploram AOP no contexto de programação funcional, com a devida comparação com nosso trabalho. Finalmente, na Seção 7.3, discorremos em torno dos principais trabalhos futuros dentro desta linha de pesquisa.

7.1 CONTRIBUIÇÕES

A principal contribuição deste trabalho é a definição de uma linguagem orientada a aspectos que disponibiliza novas abstrações para separar preocupações em programas Haskell que utilizam mônadas. AspectH, por meio de construções já bastante exploradas em outros paradigmas, oferece ao programador a possibilidade de implementar sistemas mais modulares, onde a definição de seus *crosscutting concerns* é feita de forma separada do programa base, aumentando assim a legibilidade do programa, favorecendo a manutenção e o reuso do mesmo.

A estratégia utilizada na implementação do pré-processador de AspectH, descrita no Capítulo 6, onde a introdução de construções monádicas é evitada desde o início do desenvolvimento, apesar de não ser novidade na literatura [ER04], é uma contribuição importante da pesquisa, uma vez que relatos de experimentos como este não são freqüentes. Assim, este texto preenche um espaço interessante, servindo de modelo para o desenvolvimento de outros programas que utilizem esta estratégia. Além disso, o sucesso da estratégia é reforçado com a utilização de AspectH que oferece uma visão modular para as funcionalidades impuras envolvidas com as construções monádicas.

Além disso este trabalho é evidentemente uma avaliação interessante no que diz respeito aos benefícios que AOP pode oferecer no contexto de uma linguagem funcional. Os experimentos descritos são, certamente, um indício ao paradigma funcional de que AOP oferece diversas vantagens relacionadas a modularidade, já conseguidas em outros paradigmas.

7.2 TRABALHOS RELACIONADOS

Existem na literatura alguns trabalhos que exploram AOP no contexto de linguagens funcionais. Como o leitor poderá constatar, não cabe uma comparação direta entre alguns trabalhos e AspectH, principalmente por utilizarem metodologias completamente distintas – como o uso de funcionalidades impuras sem a presença de mônadas (Seções 7.2.3 e 7.2.4) – ou por representarem estudos mais ligados à formalização de linguagens AOP (Seções 7.2.1 e 7.2.2). Um dos trabalhos descritos (Seção 7.2.5), contudo, permite uma comparação direta, devido à similaridade dos objetivos (que podem ser vistos em última análise como a definição de ferramentas para adição de ações monádicas em programas Haskell). É importante ressaltar, entretanto, que todos os trabalhos descritos resumidamente a seguir forneceram de alguma forma informações que ajudaram

a motivar a pesquisa que originou esta dissertação.

7.2.1 Mônadas como fundação teórica para AOP

Em [Meu97], Meuter sustenta que o conceito de mônada pode ser empregado para descrever formalmente a semântica de linguagens orientadas a aspectos. O autor apresenta uma pequena linguagem monádica orientada a objetos, implementada através das macros de Scheme, por meio da qual apresenta as similaridades entre programação monádica e AOP.

Nesta linguagem, uma classe possui uma superclasse e uma lista de declarações que podem ser de variáveis de instância ou de métodos. Métodos podem ser ordinários ou monádicos. Esta distinção é feita para que se possa estudar as interações entre código monádico e não-monádico. Código monádico neste contexto denota objetos monádicos, isto é, que foram injetados em uma mônada por meio da operação *return*.

O artigo expõe uma classe nesta linguagem que representa números inteiros. Esta classe possui métodos ordinários (referente a acesso a atributos e operações aritméticas) e o método monádico *fib* (que calcula a série de Fibonacci). O método *fib* é executado utilizando-se três mônadas distintas: identidade, outra que realiza *caching* dos valores calculados e uma terceira que calcula os valores da série em paralelo.

Com tais experimentos o autor sustenta seus argumentos acerca da similaridade entre programas em estilo monádico e AOP, uma vez que cada mônada utilizada provê mudanças "a nível de sistema", encapsulando os aspectos do programa (neste caso, *caching* e paralelismo) na definição da mônada. Para o autor, tanto AOP como programação monádica proporcionam a sensação de escrita de programas em camadas. A camada do topo é o código base. As outras camadas são os aspectos (ou diferentes mônadas) que auxiliam o código base a executar sua tarefa.

7.2.2 Uma teoria de aspectos

Em [WZL03], Walker et al definem a semântica de uma linguagem orientada a aspectos, MinAML, por meio da tradução de seus programas para uma linguagem núcleo (*core language*). A linguagem núcleo estende o λ-cálculo com duas novas abstrações: pontos explicitamente rotulados em um programa e *advice* de primeira classe. Os rótulos servem para delimitar as partes do programa que podem ser interceptadas. Estas abstrações são utilizadas tanto para linguagens funcionais quanto orientadas a objetos. Os autores apresentam a linguagem núcleo por meio de exemplos, provendo a semântica operacional e um sistema de tipos para ela.

A linguagem orientada a aspectos MinAML (linguagem externa) é uma linguagem funcional que permite ao programador definir *advice* estático e de segunda classe (isto é, que não pode ser manipulado como um valor). Como mencionado, sua semântica é definida por meio da tradução de seus programas para a linguagem núcleo.

A parte principal do artigo está focada no uso de aspectos no contexto de linguagens funcionais. Entretanto, os *join points* rotulados da linguagem núcleo são definidos independentemente das outras construções, podendo ser utilizados, por exemplo, no contexto de linguagens orientadas a objetos. O artigo apresenta as extensões necessárias na

linguagem externa e na linguagem núcleo para suportarem objetos, mantendo, contudo, inalterada a semântica da execução de *advice*.

7.2.3 AspectML

Dantas e Walker [DW03] apresentam a linguagem AspectML, uma extensão de Standard ML [AM91] com duas novas abstrações: pontos de fluxo de controle de primeira classe (*first-class control flow points* – CFPs) e *advice* de primeira classe (*first-class advice*).

AspectML permite ao programador declarar, armazenar e manipular CFPs, que são estruturas de dados que representam pares de pontos no fluxo de controle de um programa. Um dos pontos é o instante anterior a execução de uma função e o outro é o instante posterior. Um CFP é criado sempre que é declarada uma função em AspectML. Cada função em AspectML possui uma variável associada que representa seu CFP. Como exemplo, vejamos os CFPs correspondentes às funções *fact* e *double*, armazenados na lista *pts*:

```
1. fun fact x = if x <= 1 then 1 else fact (x-1) * x
2.
3. fun double x = 2 * x
4.
5. val pts = [$fact, $double]
```

A variável que representa o CFP de uma função é acessada prefixando-se $ ao seu identificador. Assim, *$fact* e *$double* são os CFPs das funções definidas acima. Os CFPs estão sujeitos às mesmas regras de escopo seguidas pelos identificadores de ML.

Existem duas formas de *advice* permitidos em AspectML: *before* e *after*. Como exemplo temos a seguir declarações de *advice* que executam antes e depois dos *join points* contidos na lista *pts* (vista acima):

```
1. val traceEntry =
2.   advice before <| pts as p(arg) |> @ _ =
3.    print ("enter:" ^ (label p) ^ ":"
4.           ^ (Int.toString arg))
5.
6. val traceExit =
7.   advice after <| pts as p(res) |> @ _ =
8.     print ("leave:" ^ (label p) ^ ":"
9.            ^ (Int.toString res))
```

A variável *p* (linhas 2 e 7) é ligada aos *join points* interceptados (*$fact* e *$double*) e pode ser utilizada no corpo do *advice*. As variáveis *arg* (linha 2) e *res* (linha 7) são ligadas, respectivamente, ao argumento das funções interceptadas e ao resultado produzido por elas.

Com o objetivo de proteger o código de uma função de interferência exterior (código advindo de aspectos), AspectML permite decorar suas funções com controles de acesso.

Existem quatro tipos de controle de acesso: *r* (leitura), *w* (escrita), *rw* (leitura e escrita) e *n* (nenhum). O controle de acesso é definido para os argumentos, bem como para o resultado produzido pela função. Apresentamos a seguir as funções *fact* e *double* definidas acima, agora com a especificação de controles de acesso:

```
1. fun[r,n] fact x = if x <= 0 then 1 else fact (x-1) * x
2.
3. fun[r,n] double x = 2 * x
```

A especificação de *[r,n]* define que o argumento da função pode ser lido (mas não escrito) e que o resultado da função não pode ser acessado. Com as modificações feitas nas funções *fact* e *double*, o *advice traceExit* visto acima, torna-se ilegal, uma vez que acessa o resultado das funcões selecionadas.

AspectH, se comparado com AspectML, oferece um menor controle no que diz respeito à proteção de suas funções, já que tal controle é baseado exclusivamente nas regras de visibilidade das funções dos módulos de Haskell. Entretanto, o objetivo de respeitar as propriedades de *information hiding* da linguagem é conseguido de forma satisfatória.

7.2.4 Aspectos em Scheme

Tucker e Krishnamurthi, em [TK02, TK03], apresentam uma extensão orientada a aspectos de PLT Scheme [CR+91]. Esta extensão suporta a definição de *pointcuts* e *advice*, ambos definidos como valores de primeira ordem. Sua implementação foi feita empregando-se as marcas de continuação, juntamente com as macros e o sistema de módulos de PLT Scheme.

Os *pointcuts* são definidos como funções que consomem uma lista de *join points* – que nesta linguagem são apenas chamadas de funções –, retornando um valor lógico. Um *pointcut* pode ser qualquer expressão cuja avaliação seja um predicado sobre uma lista de *join points*. Como exemplo, vejamos a seguir como é feita a definição dos operadores de *pointcut call* e *within* [1]:

```
1. (call f) = (lambda (jp) (eq? f (first jp)))
2.
3. (within f) = (lambda (jp) (and (not (empty? (rest jp)))
4.                                (eq? f (second jp))))
```

O predicado *eq?* de Scheme verifica se duas funções são idênticas. No caso de *call*, o teste feito é referente a primeira função na lista de *join points* (linha 1). Quanto a *within*, o teste se refere ao segundo elemento (linha 4).

Uma declaração de *advice* é definida como um "transformador" de funções que consome a função original e retorna a função, com a computação adicional especificada no *advice*. O único tipo de *advice* implementado é *around* (declarações de *advice* do tipo *before* e *after* podem ser implementados em termos de *around*). A forma sintática *app/prim* é utilizada dentro do escopo da expressão do *advice*, representando a execução da função

[1] O identificador *lambda* define abstrações lambda em Scheme.

interceptada (corresponde ao *proceed* de AspectH). A seguir temos um exemplo de *advice* nesta linguagem que adiciona 83 ao argumento da função interceptada:

```
(lambda (p) (lambda (a) (app/prim p(+ a 83))))
```

Para instalar *advice* em uma computação existe o operador *around*:

```
(around pcd advice body)
```

que avalia a expressão *body*, sujeita a execução de *advice* sob as condições definidas por *pcd*.

Um exemplo de sua utilização é visto a seguir:

```
1. (define (double x) (+ x x))
2.
3. (around (call double) (lambda (p)
4.                         (lambda (a)
5.                           (printf "calling double")
6.                           (app/prim p a)))
7.        (double 143))
```

O *advice* seleciona a chamada da função *double*. Ao ser executado, este programa imprime uma mensagem na saída padrão (linha 5) e retorna o valor 286.

Neste trabalho é feita a distinção entre aspectos estáticos e dinâmicos. A construção *around* define aspectos estáticos, enquanto que os dinâmicos são definidos por *fluid-around*. Uma declaração de aspecto estático se aplica a uma expressão onde quer que ela seja usada, enquanto que um aspecto dinâmico se aplica apenas no corpo da declaração do aspecto.

7.2.5 Atualização de programas funcionais

Os trabalhos apresentados em [Erw01, ER02, ER03, ER04] descrevem métodos alternativos para realização de atualizações em programas funcionais. Um dos objetivos destes trabalhos é oferecer uma visão de mais alto nível para um programa, onde as modificações realizadas neles sejam mais confiáveis e eficientes. As modificações são feitas, então, por meio de linguagens de atualização, em vez da utilização de editores de texto, onde os programas são modificados diretamente provocando mais facilmente erros diversos (de sintaxe, de tipos e lógicos). Atualizar programas por meio de tais linguagens pode evitar mais facilmente a ocorrência destes erros, tornando o processo de manutenção mais seguro, principalmente no tocante a tipos.

Destes trabalhos, apenas [ER04] pode ser comparado diretamente com o descrito neste texto, já que os demais estão centrados em programas funcionais estritos. Este trabalho apresenta um estudo aprofundado acerca do processo de transformação de programas funcionais estritos para uma forma monádica. O artigo descreve um algoritmo para realizar monadificação de programas funcionais (automatizando o processo descrito na Seção 6.5).

A monadificação é apenas um primeiro passo para adição de novas funcionalidades em um programa. Assim, o artigo apresenta também uma linguagem para atualização de programas, que adiciona ações monádicas baseado em regras de reescrita de símbolos. A despeito de não mencionar termos como aspectos ou *crosscutting concerns*, esta linguagem pode ser entendida neste contexto; e, em particular, todas as atualizações apresentadas no trabalho de Erwig e Ren podem ser realizadas (com vantagens) em AspectH. De fato, este é o trabalho na literatura relacionado de forma mais próxima com o descrito aqui. Faremos, portanto, uma comparação minuciosa, apresentando as alternativas oferecidas por AspectH às atualizações realizadas por meio desta linguagem.

A linguagem emprega regras de reescrita de símbolos para descrever a inserção de ações monádicas. A idéia de reescrita de símbolos é casar um padrão contra uma expressão ligando metavariáveis do padrão com as variáveis da expressão e assim substituir a expressão por outro padrão. A seguir vemos uma regra de reescrita que ilustra o processo descrito:

```
return (Div x y) -> if y==0 then Nothing else return (Div x y)
```

O padrão *return (Div x y)* do lado esquerdo da regra é casado com uma expressão com a mesma estrutura em um programa, que é então substituída pela expressão condicional do lado direito e as variáveis x e y substituídas pelas variáveis da expressão.

Vejamos agora como é possível implementar a transformação anterior em AspectH. Podemos expressar esta regra por meio de um *advice* do tipo *around* como visto a seguir:

```
1. around: call return &&
2.         args (Div x y)
3.    = if y==0 then Nothing else proceed (Div x y)
```

A declaração de *advice* substituirá o código de todas as chamadas à função *return* que tenham como argumento *Div x y* pela expressão definida no *advice*.

Consideremos, por exemplo, uma atualização que adicione uma ação monádica antes da expressão que define uma função. Utilizando regras de reescrita de símbolos seria necessário escrever toda a expressão que define a função do lado esquerdo da regra e, no lado direito, além da ação monádica, devemos repetir a expressão que define a função, o que pode se tornar tedioso em casos onde a expressão é longa. A seguir ilustramos a regra, onde *e* é a expressão que define a função e *e'* é a ação monádica.

```
e -> e' >>= \_ -> e
```

Em AspectH podemos expressar tal regra de forma menos prolixa, por meio de um *advice before*, não sendo necessária a repetição dos padrões que representam as expressões a serem substituídas. A seguir vemos um exemplo de *advice* que expressa a mesma transformação descrita pela regra anterior:

```
before: execution f = e'
```

Aqui *f* é o nome da função e *e'* a ação monádica.

Embora as regras de reescrita possam incorporar informações de contexto através do uso de metavariáveis em padrões, elas não podem se referir a partes do contexto que não estão sendo reescritas. Por exemplo, não é possível utilizar nas regras os parâmetros formais da função que está sendo modificada. Para superar esta limitação existe na linguagem as chamadas *regras dependentes de contexto*. Tais regras aplicam regras de reescrita a um contexto que é então executado apenas nas partes do programa limitadas a este contexto, expondo desta forma as informações de interesse.

Em AspectH informações de contexto referentes aos parâmetros de uma função podem ser expostas por meio do operador *within*. Uma combinação apropriada entre este operador e, por exemplo, *call* pode prover uma atualização que permite a inspeção dos parâmetros formais da função sendo modificada.

AspectH é uma alternativa razoável como ferramenta de transformação de código com intuito de adicionar ações monádicas, provendo um mecanismo mais eficiente para identificar *join points* em um programa, comparado com linguagens baseadas em reescrita simbólica.

7.3 TRABALHOS FUTUROS

Existem diversas linhas que podem ser propostas como extensões para trabalho descrito nesta dissertação. Algumas envolvendo diretamente o projeto desenvolvido e outras que o utilizem como base ou motivação para obtenção de resultados similares, mas usando abordagens distintas que minimizem as deficiências encontradas na abordagem original.

Iniciemos com os trabalhos futuros relacionados diretamente com a linguagem AspectH. Um dos mais importantes é, de certo, a verificação de tipos. O *Weaver* de AspectH não garante que o programa resultante do processo de combinação entre o código base e o dos aspectos tenha tipos corretos. É interessante que o pré-processador seja estendido para incluir a checagem dos tipos. Isto trará diversos benefícios à programação utilizando AspectH, uma vez que se tornará mais fácil a identificação de erros provocados pelo uso incorreto de aspectos. Além disso, considerando todas as questões envolvidas neste processo (Seção 6.1.5), é possível desvendar e esclarecer alguns pontos que o processo atual sem verificação de tipos manteve inexplorados.

Ambientes de programação (IDEs) possuem papel preponderante no desenvolvimento com linguagens orientadas a aspectos. Isto devido à falta de localidade do código definido por estas linguagens, onde tais ambientes provêem uma visualização gráfica da influência dos aspectos no código base, facilitando a identificação de onde ocorre as modificações. Um trabalho futuro interessante seria, portanto, a definição de um ambiente de desenvolvimento com suporte à linguagem AspectH. Esta definição pode ser proposta como uma extensão de alguma ferramenta existente, como por exemplo a ferramenta para *refactoring* de programas funcionais HaRe [LRT03]. É interessante também que este ambiente forneça um operador para monadificação automática de programas funcionais, motivando a introdução tardia e seletiva de mônadas em um programa, tornando o processo como um todo mais atrativo.

Outros trabalhos futuros relacionados diretamente com AspectH incluem a forma-

lização de suas sintaxe e semântica.

AspectH é um instrumento de valor na avaliação dos benefícios de AOP para programação monádica em Haskell. No entanto, como discutido anteriormente (Seções 5.8 e 6.4), a linguagem oferece algumas dificuldades no que diz respeito a especificação de como o código dos aspectos atua no programa base. A linguagem utiliza informações demasiadamente ligadas a representação textual dos programas, limitando o reuso de aspectos que de outra forma poderiam ser usados em diversos "programas base", além de tornar necessária a sincronização entre o programa base e os aspectos, já que os identificadores das funções são usados para definir a execução do código de aspectos. É interessante, portanto, o desenvolvimento de um processo onde a execução dos aspectos esteja condicionada a informações menos dependentes da representação textual dos programas, podendo assim permitir um maior reuso dos aspectos em diversas aplicações.

REFERÊNCIAS BIBLIOGRÁFICAS

[AG98] K. Arnold and J. Gosling. *The Java Programming Language*. Addison-Wesley, Second edition, 1998.

[AM91] A. W. Appel and D. B. MacQueen. Standard ML of New Jersey. In *Third International Symposium on Programming Language Implementation and Logic Programming*, volume 528, pages 1–13, August 1991.

[BH04] B. Bringert and A. Höckersten. Student Paper: HaskellDB Improved. In *Haskell Workshop*, Snowbird, Utah, USA, September 2004.

[BW88] R. Bird and P. Wadler. *Introduction to Functional Programming*. Prentice-Hall, 1988.

[CKFS01] Y. Coady, G. Kiczales, M. Feeley, and G. Smolyn. Using AspectC to improve the modularity of path-specific customization in operating system code. *SIGSOFT Softw. Eng. Notes*, 26(5):88–98, 2001.

[CR+91] W. D. Clinger, J. Rees, et al. Revised Report on the Algorithmic Language Scheme. *Lisp Pointers*, 4(3):1–55, July-September 1991.

[DW03] D. S. Dantas and D. Walker. Aspects, Information Hiding and Modularity. Technical report, Princeton University, November 2003. TR-696-04.

[ER02] M. Erwig and D. Ren. A Rule-Based Language for Programming Software Updates. In *3rd ACM SIGPLAN Workshop on Rule-Based Programming*, pages 67–77, 2002.

[ER03] M. Erwig and D. Ren. Type-Safe Update Programming. In *12th European Symposium on Programming, LNCS 2618*, pages 269–283, 2003.

[ER04] M. Erwig and D. Ren. Monadification of Functional Programs. *Science of Computer Programming*, 52(1-3):101–129, 2004.

[Erw01] M. Erwig. Programs are Abstract Data Types. In *16th IEEE Int. Conf. on Automated Software Engineering*, pages 400–403, 2001.

[FF00] R. E. Filman and D. P. Friedman. Aspect-Oriented Programming is Quantification and Obliviousness. In *Workshop on Advanced Separation of Concerns, OOPSLA*, Minneapolis, October 2000.

[Fil01] R. E. Filman. What Is Aspect-Oriented Programming, Revisited. In *Workshop on Advanced Separation of Concerns, 15th European Conference on Object-Oriented Programming*, Budapest, June 2001.

[GHJV94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1994.

[Gor88] M. Gordon. *Programming Language - Theory and its Implementation*. Prentice Hall, 1988.

[Hud00] P. Hudak. *The Haskell School of Expression: Learning Functional Programming through Multimedia*. Cambridge University Press, 2000.

[Jon94] M. P. Jones. The implementation of the Gofer functional programming system. Technical Report YALEU /DCS/RR-1030, Yale University, 1994.

[Jon95] M. P. Jones. A System of Constructor Classes: Overloading and Implicit Higher-Order Polymorphism. *Journal of Functional Programming*, 5(1):1–37, January 1995.

[Jon03] S. L. P. Jones. Haskell Language and Libraries. Technical report, Cambridge University Press, Cambridge, UK, 2003.

[JV03] D. Janzen and K. Volder. Navigating and querying code without getting lost. In *Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 178–187. ACM Press, 2003.

[KHH+01] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An Overview of AspectJ. In *European Conference on Object-Oriented Programming*, 2001.

[KLM+97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect-Oriented Programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *11th Europeen Conf. Object-Oriented Programming*, volume 1241 of *LNCS*, pages 220–242. Springer Verlag, 1997.

[KW92] D. King and P. Wadler. Combining Monads. In *Glasgow Workshop on Functional Programming, Springer Verlag Workshops in Computing Series*, July 1992.

[Lad03] R. Laddad. *AspectJ in Action*. Manning, 2003.

[LHJ95] S. Liang, P. Hudak, and M. Jones. Monad transformers and modular interpreters. In ACM, editor, *22nd Symposium on Principles of Programming Languages*, New York, NY, USA, January 1995. ACM Press.

[Lie96] K. J. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, Boston, 1996.

[LMW00] A. Lai, G. C. Murphy, and R. J. Walker. Separating Concerns with Hyper/J: An Experience Report. In *Workshop Proceedings: Multi-dimensional Separation of Concerns in Software Engineering*, pages 79–91, Limerick, Ireland, June 2000. Held at the 22nd International Conference on Software Engineering.

[LO97] K. J. Lieberherr and D. Orleans. Preventive Program Maintenance in Demeter/Java (Research Demonstration). In *International Conference on Software Engineering*, pages 604–605, Boston, MA, 1997. ACM Press.

[LOO01] K. Lieberherr, D Orleans, and J. Ovlinger. Aspect-Oriented Programming with Adaptive Methods. In *Communications of the ACM*, volume 44, pages 39–41, October 2001.

[LRT03] H. Li, C. Reinke, and S. Thompson. Tool Support for Refactoring Functional Programs. In Johan Jeuring, editor, *ACM SIGPLAN 2003 Haskell Workshop*. Association for Computing Machinery, August 2003. ISBN 1-58113-758-3.

[Meu97] W. Meuter. Monads as a theoretical foundation for AOP. In *ECOOP Workshop on Aspect-Oriented Programming*, Jyväskylä, Finland, June 1997.

[MK03] H. Masuhara and G. Kiczales. Modeling Crosscutting in Aspect-Oriented Mechanisms. In *European Conference on Object-Oriented Programming*, Darmstadt, Germany, July 2003.

[Mog89] E. Moggi. Computational Lambda-Calculus and Monads. In *Proc. of 4th Ann. IEEE Symp. on Logic in Computer Science, LICS'89 (Pacific Grove, CA, USA)*, pages 14–23. IEEE, Washington, DC, 1989.

[New04] J. Newbern. All About Monads. http://www.nomaware.com/monads/html, 2004. (último acesso 04/02/2005).

[OL01] D. Orleans and K. Lieberherr. DJ: Dynamic Adaptive Programming in Java. In *Reflection 2001: Meta-level Architectures and Separation of Crosscutting Concerns*, Kyoto, Japan, September 2001. Springer Verlag. 8 pages.

[Par72] D. L. Parnas. On the Criteria To Be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[Sun96] Sun Microsystems Inc. *Java Core Reflection. API and Specification*, October 1996.

[Tea04a] GHC Team. The Glasgow Haskell Compiler Users Guide, Version 6.0.1, 2004.

[Tea04b] SQLite Team. The SQLite Home Page. http://www.sqlite.org, 2004. (último acesso 03/12/2004).

[Tho96] S. Thompsom. *Haskell: The Craft of Functional Programming.* ADDISON-WESLEY, 1996.

[TK02] D. B. Tucker and S. Krishnamurthi. A Semantics for Pointcuts and Advice in Higher-Order Languages. Cs-02-13, Department of Computer Science, Brown University, December 2002.

[TK03] D. B. Tucker and S. Krishnamurthi. Pointcuts and Advice in higher-order languages. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development*, pages 158–167, 2003.

[TOHS99] P. L. Tarr, H Ossher, W. H. Harrison, and S. M. Sutton. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In *International Conference on Software Engineering*, pages 107–119, 1999.

[Wad90] P. Wadler. Comprehending Monads. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming, Nice*, pages 61–78, New York, NY, 1990. ACM.

[Wad92a] P. Wadler. Monads for Functional Programming. In M. Broy, editor, *Marktoberdorf Summer School on Program Design Calculi*, volume 118 of *NATO ASI Series F: Computer and systems sciences.* Springer Verlag, 1992.

[Wad92b] P. Wadler. The Essence of Functional Programming. In *19'th Symposium on Principles of Programming Languages*, pages 1–14, Albuquerque, New Mexico, January 1992. ACM Press.

[Wat90] D. A. Watt. *Programming Languages Concepts and Paradigms.* Prentice-Hall, 1990.

[WKD02] M. Wand, G. Kiczales, and C. Dutchyn. A Semantics for Advice and Dynamic Join Points in Aspect-Oriented Programming. In Gary T. Leavens and Ron Cytron, editors, *FOAL 2002 Proceedings: Foundations of Aspect-Oriented Langauges Workshop at AOSD 2002*, pages 1–8. Department of Computer Science, Iowa State University, April 2002.

[WZL03] David Walker, Steve Zdancewic, and Jay Ligatti. A Theory of Aspects. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*, August 2003.