AspectH: Uma Extensão Orientada a Aspectos de Haskell

Carlos A. R. Andrade, André L. M. Santos, Paulo H. M. Borba
Centro de Informática, Universidade Federal de Pernambuco, Av. Prof. Luiz Freire S/N,
Recife PE, Brazil 50732-970, + 55 81 32718430
email: [cara, alms, phmb]@cin.ufpe.br

Resumo

Este artigo apresenta uma extensão da linguagem de programação Haskell com o objetivo de melhorar a modularização de programas funcionais. Esta extensão, chamada AspectH, estende Haskell com conceitos de orientação a aspectos. AspectH implementa Aspect Oriented Programming (AOP) através de pointcuts e advice, como em AspectJ, e foi projetada para atuar em programas Haskell estruturados utilizando mônadas.

Abstract

This paper presents an extension of the Haskell functional programming language with the objective of improving modularization of functional programs. This extension, AspectH, extends Haskell with aspect oriented concepts. AspectH implements Aspect Oriented Programming (AOP) through pointcuts and advice, like in AspectJ, and was designed to be used in Haskell programs that use monads.

1. Introdução

Este artigo apresenta *AspectH*, uma extensão da linguagem *Haskell* [Jones 2003] com o objetivo de melhorar a modularização de programas funcionais. *AspectH* estende *Haskell* com conceitos de orientação a aspectos, implementando *Aspect Oriented Programming* (AOP) através de *pointcuts* e *advice*, como em AspectJ [Kiczales et al 2001], e foi projetada para atuar em programas *Haskell* que utilizam mônadas.

Haskell [Jones 2003] é uma linguagem funcional de propósito geral, não estrita (os parâmetros reais de uma função são avaliados apenas se e quando necessário) e puramente funcional (ausência de estado), com suporte a polimorfismo, funções de alta ordem, e tipos algébricos de dados. A linguagem possui um sistema de módulos extensível e também suporta polimorfismo *ad-hoc* (através de classes).

Após introduzir brevemente a noção de mônada (Seção 2), apresentamos as principais características de *AspectH* (Seção 3) e como é possível utilizar a linguagem para codificar exemplos clássicos de *crosscutting concerns*, como autenticação e controle de exceções, de forma modular (Seção 4). Comparamos também nosso trabalho com outras tentativas de explorar AOP em linguagens funcionais (Seção 5). O artigo encerra com nossas conclusões e perspectivas para o futuro (Seção 6).

2. Mônadas

Mônadas provêem um *framework* conveniente para linguagens funcionais suportarem efeitos encontrados em linguagens imperativas, tais como estado global, controle de exceções, *output*, e não determinismo, que não são facilmente implementados por outros mecanismos em linguagens funcionais.

Em *Haskell*, uma mônada é representada por um construtor de tipo com duas funções associadas, que é definida através da classe *Monad*.

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

Qualquer construtor de tipo *m* pode ser considerado uma mônada, uma vez que as funções *return* e >>= tenham sido definidas numa instância da classe *Monad*. Além disso, as operações da mônada devem obedecer às seguintes leis:

```
m         >>= return = m
return x  >>= f      = f x
(m >>= f) >>= g      = m >>= (\x -> f x >>= g)[1]
```

É de responsabilidade do programador definir as operações de tal forma que estas leis se apliquem.

Haskell provê a notação *do* como uma conveniência sintática para programação monádica. Expressões que utilizam *do* são traduzidas em chamadas à função >>= da seguinte forma [Jones 2003]:

```
do { e }             = e
do { e; stmts }      = e >>= \_ -> do { stmts }
do { x <- e; stmts } = e >>= \x -> do { stmts }
```

Para ilustrar o que foi dito, utilizaremos a mônada *Maybe* para capturar exceções em uma função que avalia expressões aritméticas. A seguir apresentamos o tipo de dado *Maybe* juntamente com a declaração que define *Maybe* como instância de *Monad*:

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

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

O construtor *Just* representa um estado normal associado a um valor do tipo *a*, enquanto que *Nothing* representa um estado de erro em que nenhum valor é armazenado. 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 agora o tipo de dado que representa expressões aritméticas, juntamente com a função de avaliação correspondente, extraídos de [Wadler 1995].

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


evalM :: Expr -> Maybe Int
evalM (Con x) = return x
evalM (Plus x y) = do i <- evalM x
                      j <- evalM y
                      return (i+j)

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

Os valores calculados recursivamente têm de ser ligados a variáveis (processo inverso ao da função *return*) que podem ser utilizadas em operações com inteiros. As duas

[1] Em Haskell abstrações lambda são definidas usando o caracter "\" no lugar do "λ".

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.

3. Visão Geral de AspectH

AspectH é uma extensão orientada a aspectos de *Haskell*. O modelo de *AOP* implementado pela linguagem é o de *pointcuts* e *advice* como em AspectJ: através da definição de *pointcuts* é possível interceptar certos pontos de um programa, chamados *join points*, e especificar código adicional (*advice*) que será executado lá.

AspectH atua basicamente em código monádico, embora possa, com certas restrições, modificar código escrito sem a utilização de mônadas. Nossa abordagem representa uma alternativa à estratégia de reescrita de símbolos para integrar ações a mônadas. Apresentamos mais detalhes a este respeito na seção 5.

Um aspecto é especificado da mesma maneira que um módulo em *Haskell* onde, além das declarações presentes em *Haskell*, é possível definir *advice* que será executado antes (*before*), depois (*after*) ou "ao invés" (*around*) de um *join point*. Os *join points* passíveis de interceptação são a execução ou a chamada de uma função, a definição de tipos (monádicos) e a declaração de instâncias de classe.

Os operadores de *pointcut* implementados são: *call* (intercepta chamada de funções em um módulo), *execution* (seleciona a definição de uma função), *within* (seleciona todas as chamadas de funções dentro da definição de uma função), *args* (seleciona chamadas e definições de funções que casam com o número e os padrões especificados) e *instanceDec* (seleciona a declaração de uma instância de classe). Ademais é permitida a definição de *pointcuts* mais elaborados por meio dos operadores binários *&&* (apenas os *join points* selecionados por ambos os operandos) e || (os *join points* interceptados pelos dois operandos) e o operador unário *!* (todos os *join points* que não são interceptados por seu operando).

Uma propriedade importante da programação utilizando *AspectH* é *obliviousness*: não há nada no código base em *Haskell* que indique a influência do código dos aspectos. Esta propriedade é desejável porque permite uma maior separação de preocupações durante o processo de criação do sistema [Filman 2001].

AspectH respeita as propriedades de encapsulamento de *Haskell* não sendo permitida a especificação de *advice* que influencie a semântica de funções que não foram exportadas pelo módulo (sendo invisíveis fora do mesmo).

Desenvolvemos um protótipo que implementa a linguagem através de um pré-processador fonte-a-fonte que, a partir de módulos *Haskell* e aspectos, compõe um programa final em *Haskell* refletindo as contribuições semânticas dos programas originais, processo conhecido como *weaving*.

4. Implementando Crosscutting Concerns com AspectH

Nesta seção descrevemos um exemplo da utilização de *AspectH* para implementar *crosscutting concerns* em programas funcionais monádicos. Reestruturamos um

programa, originalmente escrito apenas em *Haskell*, que implementa as funcionalidades de uma biblioteca.

Apresentamos as principais características da aplicação, observando as deficiências de modularidade que as funções do código original apresentam. Posteriormente veremos a mesma aplicação implementada com *AspectH*.

4.1 Implementação Original

A aplicação implementa as funcionalidades de uma biblioteca, gerenciando informações de usuários, livros, autores e editoras, realizando empréstimos e devolução de livros. Os usuários que fazem uso do sistema de biblioteca são classificados de acordo com os direitos de acesso: usuários com status *ADMIN* possuem acesso privilegiado ao sistema, podendo manipular diretamente o banco de dados de usuários, livros, editoras e autores, sendo autorizados a realizar empréstimo e devolução de livros; usuários com status *USER* podem apenas obter informações sobre livros, autores e editoras. Todos os usuários possuem um *login* e uma senha e precisam ser autenticados para utilizar o sistema.

Todas as funções estão localizadas no módulo *Library*, sendo que as funções exportadas (visíveis fora do módulo) verificam os direitos do usuário antes de realizar suas operações.

Para acessar banco de dados empregamos a biblioteca *HaskellDB* [Bringert e Höckersten 2004]. *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. As operações de *HaskellDB* são realizadas através da mônada *IO* de *Haskell*: todas as interações entre *Haskell* e o mundo exterior são implementadas de maneira puramente funcional por meio da mônada *IO*.

Caso haja algum problema que previna a conclusão de uma operação de *I/O*, como, por exemplo, falhas na comunicação com o banco de dados, uma exceção é lançada e, se não adequadamente tratada, o programa é encerrado.

Para evitar comportamentos anômalos como este, a mônada *IO* provê a função *catch* com o seguinte tipo:

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

A operação de *I/O* é especificada juntamente com uma função que é executada em caso de falha, evitando que o programa seja encerrado abruptamente.

Consideremos agora a função responsável pela remoção de usuários do sistema a seguir, que ilustra os sintomas de falta de modularidade que as funções do módulo *Library* apresentam.

```
rmUser :: String -> User -> IO String
rmUser me (User lg _ _ _) =
  catch
   (withDB $ \db ->
    do auth <- authenticateAdmin me
       if auth
         then do delete db users (\r -> r!login .==. c lg)
                 return "ok"
         else return "You do not have permission to delete an user.")
   (\e -> return "An error has occured...")
```

A função é definida por meio de uma chamada a função *catch*, para manipular possíveis erros de *I/O*. O primeiro argumento da função *catch* realiza a operação de remoção de usuários, fazendo primeiramente a autenticação do usuário. A função *authenticateAdmin* decide se o usuário em questão possui status *ADMIN*, retornando um valor do tipo *IO Bool*. O valor *Bool* é ligado ao identificador *auth*, que é então avaliado pelo condicional.

O segundo argumento é uma função que retorna uma mensagem de erro em caso de falha de *I/O*.

O código responsável pela autenticação do usuário e o que realiza o tratamento de exceções no sistema está espalhado por todas as funções do sistema que acessam banco de dados, representando exemplos clássicos de *crosscutting concerns*. O entendimento do código implementado desta forma é prejudicado, além de tornar mais cara a reutilização de código e a manutenção do sistema.

4.2 Implementação com AspectH

Vejamos agora a versão do código da biblioteca utilizando *AspectH*. Reescrevemos as funções do módulo *Library* retirando código referente à autenticação e controle de exceções. Implementamos estas funcionalidades em dois aspectos.

Vejamos primeiramente como fica a função *rmUser* após as mudanças feitas.

```
rmUser :: String -> String -> IO String
rmUser me (User lg _ _ _) =
    withDB $ \db ->
    do delete db users (\r -> r!login .==. c lg)
       return "ok"
```

Observe que o código da função é bem mais legível, permitindo um entendimento direto de seu objetivo.

O código referente à autenticação de usuários é implementado pelo aspecto *Authentication* a seguir.

```
aspect Authentication where

dominates ExceptionHandling

around: execution Library.rmUser && args me user
   = withDB $ \db ->
      do auth <- authenticateAdmin me
         if auth
           then proceed me user
           else return "You do not have permission to delete an user"
```

Exibimos apenas duas declarações do aspecto *Authentication*. A primeira delas, através da primitiva *dominates*, determina que o aspecto em questão tenha precedência sobre o aspecto *ExceptionHandling* durante o processo de *weaving*, isto porque a função *authenticateAdmin* pode causar um erro *I/O*. A segunda define um *advice* do tipo *around*.

O *pointcut* definido no *advice* é composto pelos operadores *execution* e *args*, através do operador *&&*: *execution Library.rmUser* intercepta a execução da função *rmUser* declarada no módulo *Library*; *args me user* seleciona todas as funções que

possuam dois argumentos. O operador *&&* determina que o *advice* atue, após o processo de *weaving*, nas funções selecionadas por ambos os operadores. A chamada à função *proceed* reinstala a computação original interceptada pelo *advice*.

A seguir temos o aspecto *ExceptionHandling* responsável pelo controle de exceções.

```
aspect ExceptionHandling where

around: execution Library.rmUser && args me user
   = catch (proceed me user)
           (\e -> return “An error has occured...”)
```

O aspecto declara um *advice* que intercepta a execução da função *rmUser* do módulo *Library*, exatamente como o aspecto *Authentication*. A expressão que define o *advice* é constituída de uma chamada à função *catch*. O primeiro argumento da função é uma chamada à função *proceed* que representa a execução da função interceptada.

A implementação da autenticação do usuário e controle de exceções em aspectos torna o código mais modular, com ganhos importantes na legibilidade das funções do código base, bem como na diminuição do esforço do programador no que diz respeito à manutenção e possível reutilização do código. Além disso, é possível ter diferentes versões do sistema com ou sem autenticação, por exemplo.

Outras funcionalidades podem ser implementadas desta forma, como *logging* das operações realizadas. Manter diferentes versões do sistema com ou sem estas funcionalidades tem um custo bem menor com o código implementado através de aspectos.

Por outro lado, uma desvantagem que pode ser apontada com esta estratégia é a dificuldade em garantir certas propriedades do programa, devido à falta de localidade do código definido em aspectos. É necessária a elaboração de novas técnicas que permitam a prova de tais propriedades na presença de aspectos.

5. Trabalhos Relacionados

[Erwig and Deling 2004] descreve um algoritmo que transforma automaticamente um grupo de funções, estruturando-as numa forma monádica. O objetivo principal desta transformação é a adição de código a estas funções. Este código é referenciado como *ações monádicas*. Após discutir diversas estratégias para adicionar ações monádicas, o artigo define a sintaxe e a semântica de uma linguagem para expressar atualizações baseadas em reescrita de símbolos.

A linguagem utiliza 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 então 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.

Existem outras regras nesta linguagem que permitem a incorporação de informações de contexto além do uso de metavariáveis em padrões, referenciadas no texto como *regras dependentes de contexto*.

Com *AspectH* é possível realizar as atualizações descritas no trabalho de Erwig e Deling. Podemos expressar a regra anterior por meio de um *advice* do tipo *around* como visto a seguir.

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

O *advice* definido 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 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 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.

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

[Tucker and Krishnamurthi 2003] expõe a implementação de aspectos em *PLT Scheme*. Eles descrevem uma pequena linguagem orientada a aspectos que foi implementada através das marcas de continuação, juntamente com macros e o sistema de módulos da linguagem *PLT Scheme*. São mostrados exemplos da utilização de *pointcuts* e *advices* de alta ordem, sendo feita a distinção entre aspectos estáticos e dinâmicos.

O trabalho, entretanto, não está limitado a um contexto puramente funcional. Os exemplos utilizados no artigo (controle de exceções e *logging* da execução de funções) dizem respeito a funcionalidades impuras, isto é, que não podem ser vistos como uma abreviação de alguma construção do lambda cálculo.

Em [Meuter 1997], Meuter sustenta que o conceito de mônada pode ser utilizado para descrever formalmente a semântica de linguagens orientadas a aspectos. É apresentada uma pequena linguagem orientada a objetos implementada utilizando-se as macros de *Scheme*, por meio da qual é demonstrada as similaridades entre programação monádica e AOP.

Este trabalho, entretanto, não possui uma relação direta com o descrito aqui e, em particular, nosso trabalho não reponde às perguntas deixadas em aberto no artigo referentes ao relacionamento entre AOP e programação monádica.

6. Conclusões e Trabalhos Futuros

Neste artigo apresentamos *AspectH* como uma alternativa para implementação de *crosscutting concerns* em programas funcionais monádicos, resultando em um código mais modular, facilitando a leitura e entendimento dos programas, bem como a manutenção e a reutilização de código. Entretanto, a utilização indisciplinada de aspectos pode tornar mais cara a verificação de propriedades de programas.

A linguagem representa uma alternativa a outras que integram ações em mônadas por meio de regras de reescrita de símbolos. *AspectH* provê um mecanismo mais expressivo e eficiente para identificar *join points* num programa. *AspectH* também representa a primeira tentativa de explorar a separação de preocupações por meio de aspectos num contexto puramente funcional.

Seria interessante que a implementação verificasse os tipos do programa *Haskell* e dos aspectos separadamente, garantindo que a combinação deles resulta em um programa com os tipos corretos. Nossa implementação não suporta esta exigência no momento, entretanto, erros de tipos são detectados automaticamente pelo compilador *Haskell*. A verificação de tipos de aspectos deverá ser realizada no futuro, principalmente com o objetivo de facilitar a localização e correção de erros causados pelo uso incorreto de aspectos. Outro trabalho futuro é a formalização da sintaxe e semântica de *AspectH*.

Referências

Jones, S. L. P. (2003) “Haskell Language and Libraries”, Cambridge University Press, Cambridge, UK.

Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm J. and Griswold, W. G. (2001) “An overview of AspectJ”. In: Workshop on Advanced Separation of Concerns, 15th ECOOP. Budapest, Hungria.

Wadler, P. (1995) “Monads for functional programming”, In: Advanced Functional Programming, vol. 925 of LNCS, Springer-Verlag.

Filman, R. E. (2001) “What Is Aspect-Oriented Programming, Revisited”. In: Workshop on Advanced Separation of Concerns, 15th ECOOP. Budapest, Hungria.

Bringert, B. and Höckersten, A.(2004) “Student Paper: HaskellDB Improved”. In: Haskell Workshop, Snowbird, Utah, USA.

Erwig, M. and Deling Ren (2004) “Monadification of Functional Programs”, Science of Computer Programming, Vol. 52, No. 1-3, 101-129.

Tucker, D. B. and Krishnamurthi, S. (2003) “Pointcuts and Advice in higher-order languages”. In: Proceedings of the 2nd International Conference on Aspect-Oriented Software Development, pages 158-167.

Meuter, W. (1997) “Monads as a theoretical foundation for AOP”. In: International Workshop on Aspect-Oriented Programming, 11th ECOOP, Jyväskylä, Finland.