Álgebra Abstrata de Tipos em Haskell

Eduardo Renesto Estanquiere

UFABC 2024.1

Resumo

Discutimos o Isomorfismo de Curry-Howard, que estabelece que programas de computador e provas matemáticas são equivalentes. Correlacionamos com técnicas de programação a nível de tipos em Haskell, chamando atenção às limitações existentes na linguagem. Propomos o projeto e implementação de um *Type Checker Plugin* para o compilador GHC que alivia algumas das limitações e facilita expressividade em programação a nível de tipos em Haskell.

Neste documento, será discutida a proposta de Projeto de Graduação em Computação de autoria de Eduardo Renesto Estanquiere, sob orientação de Prof. Dr. Emilio Francesquini e Prof. Dr. Fabrício Olivetti de França.

Introdução

Versões modernas da linguagem *Haskell* dão um bom nível de suporte a programação a nível de tipos [1]. Utilizando a extensão `DataKinds` do compilador GHC, tipos de dados algébricos são promovidos a *kinds*, onde cada construtor do tipo original se torna por si um tipo novo. Além disso, somando a extensão `TypeFamilies`, obtém-se o suporte a funções entre *kinds*, ou seja, funções que operam a nível de tipos.

Estas duas extensões, junto de várias outras fornecidas pelo GHC, permitem construções suficientemente complexas para que se escreva código Haskell com grau maior de *type safety* – mais regras de negócio podem ser descritas por relações de tipos, fazendo com que um número maior de possíveis bugs sejam detectados imediatamente em tempo de compilação.

Por outro lado, tendo boas ferramentas de programação a nível de tipos na linguagem, é natural se debruçar sobre o *Isomorfismo de Curry-Howard*, que estabelece uma equivalência entre programas e provas matemáticas, junto da interpretação de *Brouwer-Heyting-Kolmogorov* da lógica intuicionista [2], e se perguntar até que ponto tais extensões permitem utilizar a linguagem Haskell como uma linguagem de provas matemáticas.

Nesse sentido, foi feito anteriormente um trabalho [3] explorando exatamente a utilização de Haskell e suas extensões como um assistente de provas. Concluiu-se que, embora possível, dois pontos em especial se mostram como obstáculos:

1. Dada uma categoria, é possível apenas argumentar sobre um objeto específico de tal categoria, não sobre propriedades comuns a todos os objetos. Por exemplo, mostrou-se possível formalizar provas sobre o conjunto $\mathbb{N}$ dos números naturais, mas não sobre um monóide comutativo arbitrário.
2. Trabalhar com igualdade de tipos torna-se verboso, devido a necessidade de realizar *pattern-match* com o construtor `Refl` a cada nova prova utilizada para que o compilador traga suas *constraints* ao contexto.

Com tais limitações em mente, propomos o desenvolvimento de uma ferramenta que, de certa forma, as *alivie* em alguns contextos.

Objetivos

Álgebra a nível de tipos

Resolver completamente todas as limitações da linguagem de tipos em Haskell seria equivalente a transformar Haskell em uma linguagem dependentemente tipada – o que, embora existam projetos focados nesta tarefa [4], é um problema muito grande e complexo, e que foge do escopo de um projeto de graduação. Portanto, nos limitaremos a apenas *aliviar* as limitações acima apontadas em certos contextos.

A saber, o escopo proposto resume-se em *munir o compilador com resultados da álgebra abstrata*. Ao criar um *kind* novo e o munir de um número de operadores definidos como *type families*, caso seja o caso, deseja-se permitir que o desenvolvedor informe ao compilador que tais construtos formam alguma determinada estrutura algébrica, como *monóides*, *grupos*, etc.

Tendo tal informação, ao encontrar alguma igualdade de tipos, espera-se que o compilador então seja capaz de automaticamente resolver a mesma, determinando se é verdadeira ou falsa.

Sendo isso possível, aliviamos os pontos 1. porque a generalização será feita no próprio compilador, e o efeito de poder aplicar os teoremas em qualquer objeto da categoria será automático; e 2. porque o compilador resolverá automaticamente as equações que encontrar, não necessitando verificação manual de cada `Refl`.

Álgebra a nível de termos

Toda a discussão feita até o presente momento se dá a nível de tipos, e só será benéfica para código novo que dependa da nossa ferramenta.

Por outro lado, observa-se que Haskell já possui em sua biblioteca padrão classes para trabalhar com tipos que se comportam como estruturas algébricas. No entanto, tais classes são úteis apenas para generalização de código, e não exigem as propriedades que as estruturas naturalmente admitiriam num contexto matemático. Por exemplo, é possível definir uma instância de `Monoid` na qual `mappend` não é associativa e `mempty` não é uma identidade, o que contrariara a definição comum matemática de monóide.

Portanto, também propomos uma ferramenta que faça tal tipo de verificação, a nível de termos, num código pré-existente. Por hora, coloca-se como objetivo final a verificação de tais instâncias no *Hackage* inteiro.

Trabalhos similares

O projeto `ghc-typelits-natnormalize` [5] tem objetivo similar. O GHC expõe em sua API algumas formas de literais a nível de tipos, uma delas sendo números naturais. Tal projeto disponibiliza um *type-checker plugin* do compilador que normaliza igualdades entre números naturais e resolve tais equações, informando caso sejam falsas.

Semelhantemente, o projeto `type-nat-solver` [6] também exporta um plugin que resolve igualdades de tipos envolvendo tais literais. No entanto, ao invés de resolver manualmente, traduz as equações em um problema *SMT* e chama um *solver* externo para que sejam resolvidas, repassando os resultados ao compilador.

Metodologia

Assim como nos projetos descritos acima, a parte principal do projeto, que consiste na álgebra a nível de tipos, será implementado na forma de um *type-checker plugin* do GHC. A segunda parte será um *source plugin* do GHC.

Em ambas partes, será necessário resolver equações algébricas. Para tal, seguiremos a linha do projeto `type-nat-solver` e terceirizaremos a resolução para um *solver* externo. Por hora, considera-se como primeira opção o *Z3 Solver* [7].

Um plugin do GHC é, em suma, um módulo que exporta um valor do tipo `GHC.Plugins.Plugin`, e é carregado pelo compilador utilizando o argumento de linha de comando `-fplugin=<nome-do-modulo>`. Tal tipo

contém pontos de entrada para várias espécies de plugin (como os *type checker plugins* e os *source plugins*, como mencionamos acima), e cada um será chamado em um ponto diferente da compilação.

Type checker plugin

Utilizando o módulo `GHC.Tc.Types`, o GHC permite que aplicações externas se integrem aos passos de *type checking* da compilação. A interação principal se dá por meio de *Constraints* (representadas pelo tipo `Ct`), que são asserções sobre tipos que devem ser provadas ou refutadas.

Um *type-checker plugin*, considerando a versão 9.6.4 do GHC, apresenta quatro funções básicas:

```
tcPluginInit :: TcPluginM s
tcPluginSolve :: s -> TcPluginSolver
tcPluginRewrite :: s -> UniqFM TyCon TcPluginRewriter
tcPluginStop :: s -> TcPluginM ()
```

A mônada `TcPluginM` efetivamente combina as mônadas `Reader` e `IO`, além de prover algumas utilidades comuns para um *type checker*. As funções `tcPluginInit` e `tcPluginStop`, como seus nomes admitem, são utilizadas respectivamente para inicialização e deinicialização do plugin. Focaremos nas funções `tcPluginSolve` e `tcPluginRewrite`, que historicamente eram apenas uma função com comportamentos diferentes dependendo dos argumentos.

A função `tcPluginSolve` é do seguinte tipo:

```
tcPluginSolver
    :: s             --| Contexto
    -> EvBindsVar    --| Evidence Bindings
    -> [Ct]          --| Givens
    -> [Ct]          --| Wanteds
    -> TcPluginM TcPluginSolveResult
```

O propósito dessa função é provar ou refutar *constraints*. Os *givens* são constraints assumidas a serem verdadeiras, e os *wanteds* são as constraints que se desejam provar. O retorno é ou uma contradição entre *givens*, ou uma lista de constraints que foram provadamente resolvidos, possivelmente com novos constraints *wanteds* a serem provados mais à frente.

No contexto do projeto, essa função será o ponto de entrada para munirmos o compilador com os resultados de álgebra abstrata, e onde poderemos consultar o solver externo.

Semelhantemente, a função `tcPluginRewrite` tem o tipo:

```
tcPluginRewrite
    :: RewriteEnv  --| Contexto da type family
    -> [Ct]        --| Givens
    -> [TcType]    --| Argumentos da type family
    -> TcPluginM TcPluginRewriteResult
```

Seu propósito é reescrever aplicações de *type families* saturadas (isto é, sem variáveis livres), retornando coerções entre o tipo antigo e o tipo novo e possivelmente novos *wanteds* para serem utilizados mais à frente. No contexto do projeto, poderá ser aproveitada para, por exemplo, normalizar aplicações das operações algébricas para facilitar a análise e interação com o solver.

Ao final, um *type-checker plugin* prototipal pode ter como base:

```
import GHC.Plugins
import GHC.Tc.Types

pluginMain :: Plugin
pluginMain = defaultPlugin
    { tcPlugin = Just . mkTcPlugin
```

```
    }

mkTcPlugin
    :: [CommandLineOption]
    -> TcPlugin
mkTcPlugin _ =
    { tcPluginInit = myTcPluginInit
    , tcPluginSolve = myTcPluginSolve
    , tcPluginRewrite = myTcPluginRewrite
    , tcPluginStop = myTcPluginStop
    }

data PluginCtx = PluginCtx

myTcPluginInit :: TcPluginM PluginCtx
myTcPluginInit = return PluginCtx

myTcPluginSolve
    :: PluginCtx
    -> EvBindsVar
    -> [Ct]
    -> [Ct]
    -> TcPluginM TcPluginSolveResult
myTcPluginSolve ctx evs givens wanteds = do
    ...

myTcPluginRewrite
    :: RewriteEnv
    -> [Ct]
    -> [TcType]
    -> TcPluginM TcPluginRewriteResult
myTcPluginRewrite env givens args = do
    ...
```

A informação de quais kinds munidos de quais type families formam estruturas algébricas será informada pelo usuário. Idealmente, serão utilizadas anotações no código.

Source plugin

Para a segunda parte do projeto, na qual verificaremos a corretude de instâncias de estruturas algébricas a nível de termos em código pré-existente, utilizaremos um *source plugin*, que receberá do GHC a *AST* já com tipos checados. Neste ponto da compilação, teremos acesso às instâncias declaradas e poderemos argumentar sobre as definições dos elementos destacados e das funções.

Um *source plugin* é exportado de maneira similar ao *type-checker plugin* acima, apesar de consistir em apenas uma função, `typeCheckResultAction`, que recebe do compilador a *AST* e pode utilizá-la arbitrariamente dentro da mônada `TcM`.

```
import GHC.Plugins

pluginMain :: Plugin
pluginMain = defaultPlugin
    { typeCheckResultAction = myTypeCheckResultAction
    }

myTypeCheckResultAction
```

```
    :: [CommandLineOption]
    -> ModSummary
    -> TcGblEnv
    -> TcM TcGblEnv
myTypeCheckResultAction cmd mod ast = do
    ...
```

É importante mencionar que, diferente das *constraints* no caso do nível de tipos, no nível de termos não é possível argumentar sobre propriedades de funções arbitrárias – lá, os *givens* dão importante contexto, que não existe aqui. Obter o mesmo nível de informação aqui requeriria provar propriedades sobre todas as chamadas filhas de uma declaração de função numa instância de estrutura algébrica. Embora isso seja possível, como primeira ordem o escopo será manter uma tabela de operações primitivas e suas respectivas propriedades, que inicialmente será utilizada para verificar instâncias simples.

Cronograma

Tarefa	2024.1	2024.2	2024.3
Delimitação do escopo	X		
Pesquisa de trabalhos similares	X		
Implementação	X	X	X
Teste em base de código pública (parte 2/Hackage)			X
Escrita da tese		X	X

Referências

[1] E. Francesquini e F. O. de França, «Desenvolvimento Orientado a Tipos». [Em linha]. Disponível em: https://haskell.pesquisa.ufabc.edu.br/desenvolvimento-orientado-a-tipos/

[2] A. Bauer, «Five Stages of Accepting Constructive Mathematics», *Bull. Amer. Math. Soc*, vol. 54, pp. 481–498, 2016, Disponível em: https://www.ams.org/journals/bull/2017-54-03/S0273-0979-2016-01556-4/

[3] E. R. Estanquiere, «Matemática Usando Tipos - uma introdução», 2022.

[4] R. A. Eisenberg, «Dependent Types in Haskell: Theory and Practice». 2016.

[5] C. Baaij, «ghc-typelits-natnormalize». Disponível em: https://github.com/clash-lang/ghc-typelits-natnormalise

[6] I. Diatchki, «type-nat-solver». Disponível em: https://github.com/yav/type-nat-solver

[7] L. De Moura e N. Bjørner, «Z3: An efficient SMT solver», em *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, Springer, 2008, pp. 337–340.