UNIVERSIDADE DO ESTADO DE SANTA CATARINA - UDESC
CENTRO DE CIÊNCIAS TECNOLÓGICAS - CCT
MESTRADO EM COMPUTAÇÃO APLICADA

ADELAINE FRANCIELE GELAIN

INFERÊNCIA DE TIPOS NA PRESENÇA DE GADT USANDO ANTI-UNIFICAÇÃO

Dissertação submetida ao Programa de Pós-Graduação em Computação Aplicada do Centro de Ciências Tecnológicas da Universidade do Estado de Santa Catarina, para a obtenção do grau de Mestre em Computação Aplicada.

Orientador:
Prof. Dr. Cristiano Damiani Vasconcellos

JOINVILLE

2016

G314i Gelain, Adelaine Franciele
Inferência de tipos na presença de gadt usando anti-unificação/Adelaine Franciele Gelain. - 2016.
105 p. : il. ; 21 cm

Orientador: Cristiano Damiani Vasconcellos
Bibliografia: 95-97 p.
Dissertação (Mestrado) – Universidade do Estado Santa Catarina, Centro de Ciências Tecnológicas, Programa de Pós-Graduação em Computação Aplicada, Joinville, 2016.

1. Haskell (Linguagem de Programação de computador). 2.Inferência(Lógica). 3. Compiladores (Programas de computador). I.Vasconcellos, Cristiano Damiani. II. Universidade do Estado Santa Catarina. Programa de Pós-Graduação em Computação Aplicada. III. Título.

CDD: 005.133 - 23. ed.

ADELAINE FRANCIELE GELAIN

INFERÊNCIA DE TIPOS NA PRESENÇA DE GATDs USANDO ANTI-UNIFICAÇÃO

Dissertação apresentada ao Curso de Mestrado Acadêmico em Computação Aplicada como requisito parcial para obtenção do título de Mestre em Computação Aplicada na área de concentração "Ciência da Computação".

Banca Examinadora

Orientador: Prof. Dr. Cristiano Damiani Vasconcellos
CCT/UDESC

Membros

Prof. Dra. Karina Girardi Roggia
CCT/UDESC

Profa. Dra. Lucília Camarão de Figueiredo
UFOP

Joinville,SC, 29 de julho de 2016.

AGRADECIMENTOS

Gostaria de agradecer o apoio e compressão dos meus pais Avanete e Adelar durante o período do curso. Ao meu namorado Jeferson, pelo companheirismo e incentivo. Gostaria de agradecer principalmente o meu orientador Cristiano, que me possibilitou conhecer um novo mundo, que passou tardes inteiras me explicando um novo assunto, até que eu realmente entendesse. Que acreditou em mim, mesmo com toda limitação que eu tinha no começo do curso. Obrigada pela paciência e por me permitir chagar até aqui. As professoras Karina e Lucília, por aceitarem fazer parte da minha banca, suas dicas foram realmente valiosas. Agradeço também aos colegas e amigos que fiz durante o curso, aos demais professores e a CAPES, que financiou minha bolsa de estudos e me permitiu a experiência única de me dedicar exclusivamente aos estudos durante esse período.

RESUMO

Tipos de dados algébricos generalizados (GADTs) são uma extensão dos tipos de dados algébricos, disponível em linguagens funcionais como Haskell e ML, que possibilitam a generalização do resultado de uma função. Se por um lado GADTs permitem escrever programas mais concisos, por outro lado tornam a inferência de tipos mais difícil, não sendo possível, em muitos casos, determinar um tipo principal para as funções em que ocorrem GADTs. A ocorrência frequente de recursão polimórfica é outro fator complicador na inferência de tipos dessas funções. Várias abordagens foram propostas para tratar a inferência de tipos, a maioria delas porém é bastante restritiva, podendo rejeitar muitos programas potencialmente corretos ou exigindo assinatura de tipos para todos ou para a maioria dos casos. O objetivo deste trabalho é propor uma abordagem para inferência de tipos sem a necessidade de assinatura de tipos e que aceite um número maior de programas corretos. O algoritmo proposto utiliza anti-unificação para capturar a relação entre tipos, e os casos onde ocorre recursão polimórfica são tratados de forma similar a resolução de sobrecarga de símbolos em mundo fechado. Comparado com outras abordagens, acredita-se que esta proposta possibilita a inferência de um conjunto menos restrito de funções.

Palavras-chaves: gadt. inferência de tipos. tipo de dados algébricos generalizados.

ABSTRACT

Generalized algebraic data types (GADTs) are an extension of algebraic data types available in funcional languages such as Haskell and ML that make possible to have different types for the result of an expression. Programs using GADTs are very expressive, but type inference is extremely difficult due to the lost of principal type, the way type refinement is made, and due to polymorphic recursion. Many type inference approaches have been proposed, but most of those are too restritive, rejecting or asking for type signature for most of correct programs. We propose a new type inference approach that accepts more correct programs without requiring additional type signatures. Our algorithm uses anti-unification to catch the relation among types, and we treat polymorphic recusion in the same way as overloaded symbols. Compared to other approaches, we believe that we infer type for a less restrict set of functions.

Key-words: gadt. type inference. generalised algebraic data types.

LISTA DE FIGURAS

Figura 1 – Sintaxe de Tipos e Expressões 30
Figura 2 – Sistema de tipos de Damas-Milner 33
Figura 3 – Regra de especialização de tipos 33
Figura 4 – Algoritmo W: representação sob a forma de função . . 35
Figura 5 – Algoritmo W: representação sob a forma de regras de tipo 36
Figura 6 – Passos para inferência da expressão λ`f.(`λ`x.f x)`usando o Algoritmo W 37
Figura 7 – Regra que estende o sistema de Damas-Milner para suporte à recursão polimórfica 38
Figura 8 – Anti-Unificação 47
Figura 9 – Sistema de tipos ADT 55
Figura 10 – Função para avaliação de expressões, com chamadas recursivas. 101
Figura 11 – Função sem tipo principal. 102
Figura 12 – Expressão com cases aninhados. 102
Figura 13 – Função com tipo recursivo, onde o resultado das alternativas pode ser unificado. 103
Figura 14 – Função com tipo recursivo, onde não é possível unificar o tipo das alternativas. 103
Figura 15 – Função com diferentes definições para o mesmo construtor. 103
Figura 16 – Função cujo retorno não está associado ao GADT. . . . 104
Figura 17 – Função com tipo preciso: `flop1`não tem tipo principal, sendo válidos os tipos `R a` $\rightarrow$ `a`, `R a` $\rightarrow$ `Int`e `R Int` $\rightarrow$ `Int`, porém o último é considerado mais preciso por ser menos geral e evitar erros (LIN; SHEARD, 2010; VYTINIOTIS et al., 2011). 104

Figura 18 – Função com tipo generalizado: os tipos das alternativas de `flop2`variam de acordo com tipo dos construtores do GADT. 104
Figura 19 – Aplicação da função `flop2`para uma alternativa válida. 104
Figura 20 – Aplicação da função `flop2`para uma alternativa cujo construtor não foi implementado. 105
Figura 21 – Função cujo tipo da alternativa é ligado ao GADT, mas o tipo do retorno não. Novamente ocorreria o tipo `R a` $\rightarrow$ `b`, que não é válido. 105
Figura 22 – Função com expressões case aninhadas, o primeiro nível `case`está ligado ao GADT e o segundo nível não está.105

LISTA DE ABREVIATURAS E SIGLAS

GADT	Generalised Algebraic Data Types
ADT	Algebraic Data Types
CT	Constrained Types
LCG	Least Common Generalization
CS-SAT	Satisfazibilidade de Restrições
GHC	Glasgow Haskell Compiler
ML	Metalanguage, é uma linguagem de programação de propósito geral

SUMÁRIO

1 Introdução **17**

2 Fundamentos **27**

2.1 Sintaxe de Tipos e Expressões 29

2.1.1 Expressões case versus Casamento de Padrões . . 30

2.2 Sistema de Tipos de Damas-Milner 31

2.3 Recursão Polimórfica 36

2.4 Sobrecarga 41

2.5 Sistema CT 45

3 Tipos de Dados Generalizados **51**

3.1 Tipos Algébricos de Dados 51

3.2 Tipos de Dados Algébricos Generalizados 52

3.3 Sistema de Tipos para ADT e GADT 55

3.4 Abordagens para inferência de tipos para GADTs 55

3.4.1 OutsideIn 57

3.4.2 Algoritmo P e Pointwise 61

3.4.3 Chore 61

4 Inferência de Tipos Proposta **65**

4.1 Inferência de Tipo 66

4.1.1 Definição do Algoritmo 67

4.1.2 Tratamento de Recursão Polimórfica 70

4.2 Avaliação da Abordagem Proposta 70

4.2.1 Algoritmos Aceitos 71

4.2.2 Algoritmos não Aceitos 76

5 Conclusão **83**

5.1 O Sistema de Tipos é Incompleto 84

5.2 Tipo Principal 85

5.3 Refinamento de Tipo 87
5.4 Recursão Polimórfica 88
5.5 Propagação de Restrições de Tipo 89
5.6 Verificação de Erros 90
5.7 Uso de Assinatura de Tipos 91
5.8 Expressões case com tipos aninhados 92
5.9 Considerações Finais 93

REFERÊNCIAS BIBLIOGRÁFICAS 95

Anexos 99

ANEXO A Algoritmos Utilizados nos Testes 101

1 INTRODUÇÃO

Linguagens de programação modernas têm evoluído para oferecer novos recursos que possam aprimorar a forma como os programas são escritos, tornando o código mais sucinto, otimizado e mais representativo com relação ao problema que se propõe resolver. Linguagens estaticamente tipadas buscam oferecer novos recursos, aumentando a flexibilidade, sem perder a segurança. Haskell é uma linguagem estaticamente tipada, isto é, toda expressão da linguagem tem um tipo, determinado em tempo de compilação, e o compilador é capaz de inferir esse tipo sem que qualquer informação sobre tipos tenha sido fornecida pelo programador. A inferência, além de reduzir a carga de trabalho do programador, aumenta a confiabilidade do programa, já que muitos erros podem ser identificados em tempo de compilação, a partir desta relação entre programas e tipos.

Sistemas de tipos fornecem um mecanismo formal para descrever propriedades do programa, composto por *tipos* e um *conjunto de regras* que restringem o conjunto dos possíveis tipos de uma expressão. Sistemas de tipos nos quais expressões aceitam mais de um tipo utilizam *tipos polimórficos* (ou tipos quantificados), representados por *variáveis de tipo*.

Muitos sistemas de tipos permitem ao programador definir novos tipos de dados. Em linguagens funcionais recentes, como Haskell, estes tipos são chamados de *tipos algébricos de dados* (ADTs[1]), cujo nome "algébrico" se deve ao fato de que esses tipos podem ser descritos a partir de duas operações básicas: soma e produto. Na linguagem Haskell, ADTs são definidos pela cláusula `data` numa combinação de construtores e argumentos. O exemplo a seguir ilustra a definição de um ADT para representação de uma árvore binária. Essa definição consiste da palavra

[1] Na literatura a sigla ADT geralmente refere-se aos Tipos de Dados Abstratos (do inglês *Abstract Data Type*), porém para facilitar a leitura, neste trabalho utilizaremos a sigla ADT no sentido de Tipos de Dados Algébricos.

reservada `data`, seguida pelo nome do tipo de dado (`Tree`) e seu parâmetro de tipo (`a`) e, à direita do símbolo de definição "=", são especificados os construtores `No` e `Leaf` que representam, respectivamente, uma árvore com subárvores à esquerda e à direita e uma árvore vazia, ou folha.

```
data Tree a = No a (Tree a) (Tree a) | Leaf
```

Os tipos dos construtores de um ADT são inferidos automaticamente, sendo o tipo de `No :: a → Tree a → Tree a → Tree a`, e de `Leaf :: Tree a`. Tipos algébricos são homogêneos: na construção de um dado, cada parâmetro de tipo definido no ADT pode ser instanciado para um único tipo. Por exemplo, as seguintes expressões representam, respectivamente, uma árvore contendo valores inteiros e outra contendo caracteres:

```
No 2 (No 1 Leaf Leaf) Leaf :: Tree Int
No 'a' Leaf (No 'b' Leaf Leaf) :: Tree Char
```

O tipo de dado `Tree a` é um tipo recursivo, uma vez que o construtor `No` recebe dados do tipo `Tree a` como parâmetros. A definição do tipo `Tree a` estende o conjunto de tipos definidos por padrão na linguagem e que podem ser usados em programas. A função `insert` é definida a seguir utilizando o mecanismo de casamento de padrão disponível em Haskell: cada alternativa na definição da função corresponde a um dos construtores do tipo do argumento da função – `Tree a`. A alternativa correspondente a uma árvore vazia (construtor `Leaf`) cria uma árvore tendo como raiz o elemento `x` passado como parâmetro. A alternativa correspondente a uma árvore não vazia (construtor `No`), inclui o elemento `x` passado como parâmetro na subárvore à esquerda ou à direita, conforme `x` seja menor ou maior que o valor contido na raiz da árvore, respectivamente.

```
insert :: Ord a ⇒ a → Tree a → Tree a
insert x (Leaf) = No x Leaf Leaf
insert x t@(No a l r) | (x > a) = No a l (insert x r)
                      | (x < a) = No a (insert x l) r
                      | otherwise = t
```

Tanto na definição do tipo de dado `Tree a`, quanto na definição da função `insert`, são utilizadas variáveis de tipo para representar os vários tipos possíveis para estas definições. No caso da função `insert`, estes tipos se restringem aos tipos que sejam instâncias da classe `Ord`[2]. Note que a anotação do tipo da função `insert` poderia ser omitida, pois o mecanismo de inferência de tipos de Haskell é capaz de inferir o tipo `insert` sem que nenhuma informação de tipo seja fornecida pelo programador.

ADTs são bastante flexíveis e permitem diferentes construtores de dados, mas o tipo representado pela variável não pode ser instanciado com tipos distintos em uma declaração. Quando é necessário manipular dados cuja variável pode ser instanciada com tipos distintos de acordo com o construtor utilizado, como é o caso da função `eval` apresentada a seguir (exemplo adaptado de (JONES; WASHBURN; WEIRICH, 2004)), é necessário encapsular o resultado da expressão num dado intermediário. A função `eval` recebe como argumento um valor do tipo `Term`, definido abaixo, que representa uma expressão a ser avaliada, e retorna o valor obtido pela avaliação dessa expressão. Esse valor é encapsulado como um dado do tipo `Val`, em que cada construtor é uma etiqueta para cada um dos possíveis tipos (inteiro, booleano ou uma dupla) do valor que a avaliação de `Term` pode resultar:

[2] Na classe `Ord` são declaradas definições sobrecarregadas para a função `compare` que é usada na declaração dos operadores: `(<)`, `(>)`, `(<=)` e `(>=)`.

```
data Term  = Lit Int    | Inc Term
           | IsZ Term   | If Term Term Term
           | Pair Term Term

data Val = VInt Int | VBool Bool | VPr Val Val

eval :: Term → Val
eval (Lit i)    = VInt i
eval (Inc t)    = case eval t of
                  VInt i → VInt (i + 1)
eval (IsZ t)    = case eval t of
                  VInt i → VBool (i == 0)
eval (If b t e)= case eval b of
                  VBool True → eval t
                  VBool False → eval e
eval (Pair l r) = VPr (eval l) (eval r)
```

Esta construção tem dois problemas principais:

- Requerer um dado intermediário (de tipo `Val`) para encapsular o resultado, gerando um *overhead* na implementação.
- Permitir que construções incorretas sejam aceitas: por exemplo, a expressão `(Inc (IsZ (Lit 0)))` é inválida, pois o construtor `Inc` manipula inteiros e está aplicando o resultado de `IsZ`, que é booleano, mas é aceita num contexto onde a avaliação de `Term` é representada conforme acima.

Para contornar estes problemas, foi proposto o *tipo de dado algébrico generalizado* (*generalised algebraic data type* - GADT), que estende tipos de dados algébricos e permite que o parâmetro de tipo de um tipo de dado algébrico varie de acordo com sua estrutura (JONES et al., 2006).

A função `eval` poderia ser escrita utilizando GADT, como mostrado a seguir:

```
data Term a where
     Lit   :: Int → Term Int
     Inc   :: Term Int → Term Int
     IsZ   :: Term Int → Term Bool
     If    :: Term Bool → Term a → Term a → Term a
     Pair  :: Term a → Term b → Term (a,b)

eval :: Term a → a
eval (Lit i)      = i
eval (Inc t)      = 1 + eval t
eval (IsZ i)      = 0 == eval i
eval (If b c d)   = if eval b then eval c
                    else eval d
eval (Pair a b)   = (eval a, eval b)
```

Na definição de um GADT, o tipo de cada um dos construtores (`Lit`, `Inc`, `IsZ`, `If` e `Pair`) é definido pelo programador, sendo que cada construtor pode instanciar o parâmetro de tipo do GADT com um tipo distinto. Por exemplo, `Lit` constrói um dado de tipo `Term Int`, já `IsZ` constrói um dado de tipo `Term Bool`.

Com esta definição de `Term` (utilizando GADT), a expressão `(Inc (IsZ (Lit 0)))` será rejeitada pelo compilador, pois o tipo de cada construtor é explícito, e o tipo inferido para `IsZ` (`Term Bool`) é diferente do tipo do parâmetro esperado por `Inc` (`Term Int`). Com isso, ao contrário do algoritmo escrito utilizando ADTs, ao utilizar os GADTs temos dois principais benefícios:

- No tipo de `eval :: Term a → a`, o tipo do resultado `a` é o mesmo do argumento (`Term a`), podendo ser instanciado para tipos distintos (definidos no GADT); sendo assim não é necessário encapsular o resultado num dado intermediário.

- Apenas expressões corretas são aceitas pelo compilador, não sendo possível que expressões incorretas sejam aceitas, o que ocasionaria erro em tempo de execução.

Algumas estruturas de dados complexas possuem regras que não devem ser violadas para garantir uma determinada propriedade. Por exemplo, para árvores vermelha e preta são definidas as seguintes regras invariantes com objetivo de garantir seu balanceamento: 1) todo nó deve ser vermelho ou preto; 2) a raiz da árvore deve ser preta; 3) as folhas devem ser pretas; 4) nós vermelhos têm filhos pretos; 5) para todos os nós internos, cada caminho daquele nó para a folha descendente contém o mesmo número de nós pretos.

Utilizando ADTs, árvores vermelha e preta seriam definidas de forma semelhante ao que é apresentado abaixo:

```
data Color = Red | Black
data Tree a = Tree Color a (Tree a) (Tree a) | Leaf
```

Novamente, este tipo de dado permite criar estruturas que não respeitam as regras invariantes. Como exemplo, uma árvore com folhas vermelhas: `Tree (Red, 2, Tree (Red, 1, Leaf, Leaf), Tree (Red, 3, Leaf, Leaf))`. Utilizando GADTs, o dado poderia ser reescrito da seguinte forma:

```
data Red
data Black
```

```
data Tree a where
  Root :: Node Black a → Tree a

data Node c a where
  Leaf  :: Node Black a
  NodeB :: Node c a →   Node c a → Node Black a
  NodeR :: Node Black a → Node Black a → Node Red a
```

Desta forma muitas construções incorretas poderiam ser rejeitadas pelo sistema de tipos. A definição de um GADT permite que o programador descreva algumas propriedades da estrutura de dados na declaração do tipo, sendo essas propriedades verificadas pelo compilador; permite também que expressões como `eval` sejam implementadas de forma mais natural e mais eficiente. De forma sumarizada, as principais características que diferem ADTs de GADTs são listadas a seguir.

ADTs:

- Permitem a criação de tipos homogêneos;
- O tipo dos construtores é inferido;
- Pode aceitar expressões incorretas;
- Expressões como `eval` são menos eficientes quando descritas utilizando ADTs.

GADTs:

- Permitem a generalização do resultado;
- Tipo dos construtores é assinado;
- Erros em tempo de execução podem ser evitados;

- Permite que invariantes sejam definidas na própria declaração do dado, dessa forma o sistema de tipos garante a não violação;
- Expressões como `eval` são implementadas de forma mais natural e eficiente.

No entanto as características dos GADTs tornam bastante complexo o processo de inferência de tipos. No processo de inferência de tipos, o sistema de tipos deve aceitar apenas programas corretos e rejeitar programas incorretos, muito embora possam ser rejeitados programas potencialmente corretos para garantir a consistência do sistema de tipos. Em linguagens estaticamente tipadas, um programa é considerado correto se existe pelo menos um tipo que representa cada expressão, garantindo portanto, que não ocorrem erros de tipo durante a execução do programa. Os programas corretos podem ser identificados por meio da verificação de tipos (*type checking*) ou pela inferência de tipos (*type inference*) (LIN, 2010).

A inferência de tipos reduz a carga de trabalho do programador, por aceitar programas que não tenham assinatura de tipos. Entretanto, a análise semântica torna-se muito mais difícil, se comparada com o caso de verificação de tipos, pois é necessário avaliar a corretude do programa sem que nenhuma informação de tipos seja fornecida explicitamente. A inferência de tipo para funções onde ocorrem GADTs é particularmente difícil, devido à variação do tipo do retorno, o que influi na forma como será feito o refinamento de tipos das alternativas (não é possível resolver por unificação), no tratamento de chamadas recursivas (que envolve resolver recursão polimórfica), além de resultar na perda da propriedade de tipo principal em alguns casos.

Várias abordagens foram propostas com o intuito de possibilitar o uso de GADTs, a maioria dessas abordagens exige assinatura de tipos. Trabalhos mais recentes vêm propondo mais que isso, possibilitar o uso de GADTs sem perder recursos como inferência de tipo, importante em Haskell. GADTs são suportados pela versão 7.10.1 do GHC

(TEAM, 2015) como descrito em (JONES et al., 2006), exigindo, na maioria dos casos, assinatura de tipos. Por outro lado, trabalhos como o de Chen e Erwig (2016), Vytiniotis et al. (2011), Lin e Sheard (2010), Sulzmann, Schrijvers e Stuckey (2008), Stuckey e Sulzmann (2005) trazem mais flexibilidade no uso de GADTs, porém ainda restringindo o conjunto de programas aceitos sem assinatura, devido a questões envolvendo a principalidade de tipos em funções, o tratamento de recursão polimórfica (que ocorre com frequência), ou a propagação de tipos em escopos aninhados. Neste trabalho é proposta uma abordagem alternativa, com o objetivo de possibilitar o uso de GADTs de forma mais natural e próxima do proposto no sistema de Damas Milner (mais detalhes na Seção 2.2). Sendo assim, as principais contribuições deste trabalho são:

- Apresentar um novo algoritmo capaz de inferir o tipo de funções onde ocorrem GADTs para um conjunto menos restrito de programas, assim como uma implementação desse algoritmo.

- Sugerir uma proposta alternativa para tratamento de recursão polimórfica. Esses casos são tratados de forma similar a símbolos sobrecarregados no sistema CT (mais detalhes na Seção 2.5). Cada alternativa de uma expressão `case` envolvendo GADTs é tratada como uma instância do símbolo sobrecarregado, que é então tratado como um problema de verificação da satisfazibilidade das restrições. Desta forma podemos delimitar o conjunto de expressões em que é possível garantir decidibilidade da inferência, conforme descrito por (RIBEIRO; CAMARAO; FIGUEIREDO, 2013).

- A solução proposta adiciona pouca complexidade ao sistema de tipos existente, basicamente adicionando novas regras para inferência de expressões CASE, as demais regras seguem a mesma premissa de inferência de expressões que não envolvem GADT.

Alguns conceitos são necessários para o entendimento deste trabalho, no Capítulo 2 são abordados conceitos relacionados ao sistema de tipos (de Damas-Milner e CT) e recursos da linguagem (recursão polimórfica e sobrecarga) importantes na inferência de tipos, especialmente de funções que utilizam GADTs. Os trabalhos relacionados são apresentados no Capítulo 3. No Capítulo 4 o algoritmo proposto é abordado, bem como uma análise dos algoritmos aceitos ou não, seja pela abordagem proposta, quanto pelas abordagens relacionadas.

2 FUNDAMENTOS

Sistemas de tipos fornecem um mecanismo formal para descrever propriedades de um programa, e são compostos por *tipos* e um *conjunto de regras* para deduzir o tipo de expressões de programas. Alguns exemplos de tipos são listados abaixo:

```
Int          -- Inteiros
Char         -- Cacteres
[Char]       -- Lista de caracteres
Int → Bool   -- Função que mapeia um valor inteiro
                em um booleano
```

Regras de tipo definem uma relação entre tipos e programas, ou seja, o tipo de um programa num sistema de tipos. Esta relação entre programa e tipo em Haskell é especificado por meio do operador (::). A expressão `7 + 6 :: Int` é válida e pode ser lida como “o programa `7 + 6` tem tipo `Int`”[1].

Alguns podem ver tipos apenas como uma forma de documentação, como comentários. Porém, ao contrário dos comentários que têm forma livre e podem fornecer uma informação ambígua ou errada, os tipos são precisos e consistentes, pois têm uma sintaxe definida formalmente (LIN, 2010). Além disso, tipos auxiliam no processo de detecção de erros, na definição de abstrações e permitem ao programador restringir os valores de uma expressão, como no exemplo abaixo, onde `a → [a]` e `Int → [Int]` são tipos aceitos para a expressão, porém o primeiro possibilita o uso da função `\x → [x]` para um conjunto maior de tipos de argumentos.

[1] O tipo da expressão `7 + 6` é na verdade `Num a ⇒ a`, pois o símbolo de adição é sobrecarregado para diferentes tipos numéricos, instâncias da classe `Num`. É usado o tipo `Int` apenas para facilitar a compressão.

```
\ x → [x] :: a → [a]
\ x → [x] :: Int → [Int]
```

Sistemas de tipos com suporte a polimorfismo paramétrico geralmente incluem variáveis de tipo para representar tipos polimórficos. Uma variável de tipo `a` representa um tipo qualquer, sendo o tipo `Int → [Int]`, do exemplo anterior, uma instância do tipo `a → [a]` (ou $\forall$`a.a → [a]`, explicitando o quantificador universal como em (DAMAS; MILNER, 1982)).

Um sistema de tipos permite que muitos dos erros de implementação sejam detectados em tempo de compilação, desde que o sistema seja consistente. Na maioria das linguagens tipos apenas são avaliados mediante assinatura de tipos das variáveis usadas no programa. Outras linguagens, possuem um sistema de tipos capaz de determinar o tipo de expressões sem que o programador tenha que fornecer qualquer informação de tipo (apesar de possibilitar). É o que ocorre em Haskell, no processo de inferência de tipos.

A inferência de tipos é um importante passo que linguagens funcionais têm dado no sentido de aumentar a flexibilidade e o poder de expressão de linguagens estaticamente tipadas, como ocorre em linguagens dinamicamente tipadas, porém sem perder a segurança que a tipagem estática traz. O algoritmo de inferência deve ser capaz de aceitar apenas programas bem tipados, e rejeitar programas mal tipados, ou potencialmente incorretos, em tempo de compilação. Geralmente o processo de inferência de tipos combina esses dois processos, de *verificação de tipos* e *inferência de tipos*, este segundo sem a necessidade de informação de tipo.

Se por um lado a inferência de tipos reduz a carga de trabalho do programador, que não precisa fornecer assinatura de tipo para toda expressão, por outro lado torna a fase de análise semântica do compilador muito mais difícil, pois precisa decidir se o programa é bem tipado sem saber o tipo esperado pelo programa. A inferência de funções envolvendo

GADTs é ainda mais difícil, devido às restrições de tipo impostas ao construtor na definição do dado, e algumas particularidades da construção destes dados, que serão discutidas no Capítulo 3. Devido à dificuldade na inferência de tipos de expressões envolvendo GADTs, a assinatura de tipos é tida como uma alternativa para casos onde o compilador rejeita programas corretos indevidamente, ou casos cuja inferência é muito difícil (ou indecidível), como ocorre com a inferência de tipos de funções envolvendo recursão polimórfica.

No decorrer deste capítulo, é apresentada uma breve revisão sobre o sistema de tipos de Damas-Milner (Seção 2.2), criado originalmente para ML e estendido para as linguagens funcionais modernas como Haskell. Neste capítulo também é apresentada uma revisão sobre o sistema CT (Seção 2.5), que assim como Haskell estende o sistema de Damas-Milner adicionando suporte a sobrecarga, e cujas ideias são utilizadas neste trabalho para o tratamento de inferência de tipos nos casos que ocorre recursão polimórfica.

2.1 Sintaxe de Tipos e Expressões

Nesta seção é apresentada a notação que será utilizada no decorrer deste trabalho. A Figura 1 contém a sintaxe de tipos e expressões. Para facilitar, e seguindo uma prática comum, não serão considerados *kinds* no tipo de expressões.

Uma substituição é uma função de variáveis de tipo em expressões de tipo simples. $\mathtt{S}(\tau)$ denota a substituição $\mathtt{S}$ aplicada sobre o tipo τ, que também pode ser escrita como $\mathtt{S}\tau$. Uma substituição é usualmente escrita como um mapeamento finito $[\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n]$, também representada na forma $[\overline{\alpha \mapsto \tau}]$ ou $[\alpha_i \mapsto \tau_i]^{i=1..n}$.

A aplicação de substituição é sobrecarregada para restrições de tipo, conjuntos de restrições e conjuntos de tipos. O símbolo $\circ$ representa composição de função; a composição $\mathtt{S} \circ \mathtt{S'}$ também é usada na forma $\mathtt{SS'}$, e $\mathtt{dom}(\mathtt{S}) = \{\alpha \mid \ \mathtt{S}(\alpha) \neq \alpha\}$.

Meta-Variáveis:	a,b,x,y		
Variáveis de tipo:	α, β		
Construtores de tipo:	T		
Construtores de dado:	C,D		
Sequência:	$a_1,\ldots,a_n$ **ou** $\overline{a}^n$ **ou** $\overline{a}$, **onde** $n \geq 0$		
Expressões:	e	:=	$x \mid C \mid \lambda x.e \mid e\ e' \mid$ let x = e in e' $\mid$ case e of $\overline{p \rightarrow e'}$
Tipos Simples:	τ, ρ, υ	::=	$\alpha\ \overline{\tau} \mid T\ \overline{\tau}$
Tipos Polimórficos	σ	::=	$\tau \mid \forall\overline{\alpha}.\sigma$
Padrões (patterns):	p	::=	$x \mid C\ \overline{p}$
Restrições:	κ	::=	$\{\overline{x : \tau}\}$
Contextos de tipo:	Γ	::=	$\varepsilon \mid \Gamma, x{:}\sigma$

Figura 1: Sintaxe de Tipos e Expressões

Operações sobre contextos de tipos são definidas a seguir:

$$
\begin{aligned}
\Gamma(x) &= \{\sigma \mid x : \sigma \in \Gamma\} \\
\Gamma, x : \sigma &= (\Gamma - \{x : \sigma \mid \sigma \in \Gamma(x)\}) \cup \{x : \sigma\}
\end{aligned}
$$

A notação $\mathtt{ftv}(\sigma)$ denota o conjunto de variáveis de tipo livres em σ, ou seja, variáveis de tipo que não estão ligadas ao quantificador universal $\forall$. De forma similar ftv (Γ) denota o conjunto de variáveis de tipo livres em Γ.

2.1.1 Expressões case versus Casamento de Padrões

Funções que avaliam alternativas para diferentes construtores de um tipo de dado podem ser escritas utilizando expressões `case` ou casamento de padrões. Casamento de padrões é um açúcar sintático para expressões `case` que Haskell disponibiliza e que costuma facilitar a escrita e leitura dos programas. Assim, a função `eval` (descrita no capítulo 1), é equivalente à implementação abaixo utilizando `case`:

```
eval :: Term a → a
eval e = case e of
           Lit i     → i
           Inc t     → 1 + eval t
           IsZ i     → 0 == eval i
           If b c d  → if eval b then eval c
                       else eval d
           Pair a b  → (eval a, eval b)
```

Na definição acima, `e` representa o tipo que está sendo avaliado (tipo *scrutinee*), as alternativas possíveis são descritas após o `of`, cada alternativa corresponde a um padrão possível para `e`; e do lado direito da seta é definida a expressão que especifica o retorno quando o valor de `e` casa com o padrão.

No decorrer deste trabalho ambas notações podem ser utilizadas, dependendo do que se pretende expor.

2.2 Sistema de Tipos de Damas-Milner

O sistema de Damas e Milner (1982), proposto originalmente para a linguagem ML e adotado pela maioria das linguagens funcionais, descreve um algoritmo capaz de inferir o tipo menos geral capaz de descrever todos os tipos possíveis que uma expressão pode assumir (tipo principal).

Esse sistema, apresentado na Figura 2, descreve o conjunto de expressões bem tipadas, considerando a sintaxe definida na Figura 1. Um julgamento de tipo nesse sistema tem forma $\Gamma \vdash \mathtt{e} : \sigma$, que pode ser lido "*a expressão* `e` tem tipo σ no contexto Γ". Um contexto de tipos Γ consiste em uma lista de pares $\mathtt{x}:\sigma$, denominadas "suposições de tipo". Γ

armazena informação de tipo das variáveis declaradas, permitindo avaliar se uma expressão é bem tipada a partir dos tipos das variáveis livres na expressão. Por exemplo, a expressão `x + 1` é bem tipada se o operador `(+)` tem tipo `Int` $\rightarrow$ `Int` $\rightarrow$ `Int` e `x` também tiver sido definido como inteiro. $\sigma \leq \sigma'$ especifica uma relação de ordem entre tipos, que indica que o tipo polimórfico σ é *mais geral* que o tipo σ'. Essa relação é definida pela regra de especialização de tipos apresentada na Figura 3. A função `close` representa a quantificação de variáveis de tipo, e é definido como close$(\Gamma, \tau) = \forall \alpha_1 ... \alpha_n . \tau$, onde $\alpha_1 ... \alpha_n$ são variáveis de tipo que ocorrem em τ, mas não em Γ, $\forall \overline{\alpha}.\tau$ representa abreviadamente close(Γ, τ).

O sistema de tipos apresentado na Figura 2 consiste de um conjunto de regras sensíveis ao contexto, que define o tipo de cada expressão, garantindo que cada operação tenha operandos adequados, ou seja, com tipos válidos. Cada regra de tipo representa uma condição no sistema de tipos, onde:

VAR representa a relação necessária para atribuição de tipos à uma variável em um contexto de tipos: `x` tem tipo σ no contexto Γ, se houver uma instância de `x`, com esse tipo, no contexto.

INST indica que σ' é uma instância de tipo para a expressão `e` no contexto Γ, se `e` tiver tipo σ no contexto Γ e σ' for uma instância de σ.

GEN descreve a definição de variáveis com tipo quantificado, sendo que a expressão `e` tem tipo $\forall \alpha . \sigma$ se `e` tiver tipo σ e α não for uma variável livre no contexto Γ.

APP representa a aplicação de funções, sendo que a função `e` aplicada ao argumento `e'` tem tipo τ', se `e'` é compatível com o tipo do argumento esperado por `e`, ou seja, o domínio da função deve ter o mesmo tipo τ que o argumento.

ABS descreve lambda abstração, λ`x` representa o argumento para a função `e`, sendo τ o tipo de `x` e τ' o tipo de `e`, o tipo de λ`x.e` é $\tau \rightarrow \tau$'.

LETREC-M introduz uma nova definição para `x` no contexto em que `e'`

$$\frac{\mathtt{x} : \sigma \in \Gamma}{\Gamma \vdash \mathtt{x} : \sigma} \quad \text{(VAR)}$$

$$\frac{\Gamma \vdash \mathtt{e} : \sigma \qquad \sigma \leq \sigma\text{'}}{\Gamma \vdash \mathtt{e} : \sigma\text{'}} \quad \text{(INST)}$$

$$\frac{\Gamma \vdash \mathtt{e} : \sigma \qquad \alpha \notin \mathtt{ftv}(\Gamma)}{\Gamma \vdash \mathtt{e} : \forall\alpha.\sigma} \quad \text{(GEN)}$$

$$\frac{\Gamma \vdash \mathtt{e} : \tau \to \tau\text{'} \qquad \Gamma \vdash \mathtt{e'} : \tau}{\Gamma \vdash (\mathtt{e\ e'}) : \tau\text{'}} \quad \text{(APP)}$$

$$\frac{\Gamma, \mathtt{x} : \tau' \vdash \mathtt{e} : \tau}{\Gamma \vdash (\lambda\mathtt{x.e}) : \tau\text{'} \to \tau} \quad \text{(ABS)}$$

$$\frac{\Gamma, \mathtt{x} : \tau \vdash \mathtt{e} : \tau \quad \Gamma, \mathtt{x} : \sigma \vdash \mathtt{e'} : \tau\text{'} \quad \sigma = \mathtt{close}(\Gamma, \tau)}{\Gamma \vdash (\mathtt{let\ x\ } = \mathtt{e\ in\ e'}) : \tau\text{'}} \quad \text{(LETREC-M)}$$

Figura 2: Sistema de tipos de Damas-Milner

$$\frac{\tau\text{'} = [\alpha_i \mapsto \tau_i]\tau \qquad \beta_i \notin \mathtt{ftv}(\forall\alpha_1 \ldots \alpha_n.\tau)}{\forall\alpha_i \ldots \alpha_n.\tau \leq \forall\beta_i \ldots \beta_n.\tau\text{'}}$$

Figura 3: Regra de especialização de tipos

terá seu tipo inferido. O tipo das ocorrências recursivas de `x` em `e` são tradados como tipos monomórficos.

As regras descritas na Figura 2 impõe um conjunto de condições para dedução do tipo de expressões, porém não fornecem um método para inferência de tipos efetivamente. Como pode ser observado, existem casos onde pode haver mais de uma regra para a mesma fórmula (como ocorre para INST e GEN). Para inferência de tipos neste sistema é utilizado o *Algoritmo W* (DAMAS; MILNER, 1982), cuja definição é baseada no uso de unificações.

Pode-se dizer que dois tipos $\mathtt{t}_1$ e $\mathtt{t}_2$ podem ser unificados se existe uma substituição `S` que aplicada a estes tipos, torna-os iguais, ou seja:

$$\mathtt{S(t_1)\ =\ S(t_2)}$$

Por exemplo, é possível afirmar que os tipos $\mathtt{a} \rightarrow \mathtt{Int}$ e $\mathtt{Bool} \rightarrow \mathtt{b}$ são unificáveis pois existe uma substituição $\mathtt{S = [a} \mapsto \mathtt{Bool,\ b} \mapsto \mathtt{Int]}$ que, aplicada a ambos os tipos, tem o mesmo resultado: $\mathtt{Bool} \mapsto \mathtt{Int}$. Porém, os tipos $\mathtt{Int} \rightarrow \mathtt{a}$ e $\mathtt{Bool} \rightarrow \mathtt{b}$ não são unificáveis, pois não existe uma substituição que aplicada a ambos retorne o mesmo tipo, pois não existe um tipo que represente `Int` e `Bool`[2].

O Algoritmo W, apresentado na Figura 4, tem como entrada um par, composto por um contexto de tipos Γ e uma expressão `e`, e retorna o tipo principal da expressão τ e uma substituição `S`. É indicado um erro em casos onde não seja possível definir um tipo principal para a expressão.

Na Figura 4, β é uma variável livre, adicionada ao contexto quando uma nova variável de tipo é necessária, por exemplo, a função identidade `f x = x` pode ser reescrita como $\lambda\mathtt{x.x}$, onde `x` tem tipo β, adicionado pelo algoritmo. unify(τ_1, τ_2) representa o unificador mais geral para o par de expressões de tipo. Um unificador $\mathtt{S_g}$ é o unificador mais geral se, para qualquer outro unificador `S`, existe uma substituição `S'` tal que $\mathtt{S'} \circ \mathtt{S_g} = \mathtt{S}$.

O Algoritmo W apresentado na Figura 4 sob a forma de função, pode ser também descrito sob a forma de regras de inferência, tal como na Figura 5. As funções `close` e `unify` foram mantidas para facilitar a leitura.

2 Algoritmos eficientes de unificação - que podem encontrar o unificador mais geral para um par de tipos em tempo linear - foram propostos por (PATERSON; WEGMAN, 1976; MARTELLI; MONTANARI, 1982).

$$
\begin{array}{l}
W(\Gamma, \mathtt{x}) \;\mathtt{=} \\
\qquad \mathtt{Se}\ \Gamma(\mathtt{x})\ \mathtt{=}\ \forall\alpha_1...\alpha_2.\tau\ \ \mathtt{ent\tilde{a}o}\ \ (\overline{[\alpha \mapsto \beta]}\tau, \mathtt{Id}) \\
\qquad \mathtt{senao\ Falha} \\
\\
W(\Gamma, \mathtt{e\,e'}) \;\mathtt{=} \\
\qquad \mathtt{let}\ (\tau, \mathtt{S}_1)\ \mathtt{=}\ W(\Gamma, \mathtt{e}) \\
\qquad\quad\ \ (\tau', \mathtt{S}_2)\ \mathtt{=}\ W(\mathtt{S}_1\Gamma, \mathtt{e'}) \\
\qquad\quad\ \ \mathtt{S}\ \mathtt{=}\ \mathtt{unify}\ (\mathtt{S}_2\tau, \tau' \to \beta)\ \mathtt{onde}\ \beta\ \mathtt{fresh} \\
\qquad \mathtt{in}\ (\mathtt{S}\beta, \mathtt{S} \circ \mathtt{S}_2 \circ \mathtt{S}_1) \\
\\
W(\Gamma, \lambda\mathtt{x.e}) \;\mathtt{=} \\
\qquad \mathtt{let}\ (\tau, \mathtt{S})\ \mathtt{=}\ W((\Gamma, \{\mathtt{x} : \beta\}), \mathtt{e}) \\
\qquad \mathtt{in}\ (\mathtt{S}(\beta \to \tau), \mathtt{S}) \\
\\
W(\Gamma, \mathtt{let\ x\ =\ e\ in\ e'}) \;\mathtt{=} \\
\qquad \mathtt{let}\ (\tau, \mathtt{S}_1)\ \mathtt{=}\ W(\Gamma, \mathtt{e}) \\
\qquad\quad\ \ (\tau\mathtt{'}, \mathtt{S}_2)\ \mathtt{=}\ W(\mathtt{S}_1(\Gamma, \{\mathtt{x} : \mathtt{close}(\mathtt{S}_1\Gamma, \tau)\}), \mathtt{e'}) \\
\qquad \mathtt{in}(\tau\mathtt{'}, \mathtt{S}_1 \circ \mathtt{S}_2)
\end{array}
$$

Figura 4: Algoritmo W: representação sob a forma de função

Como exemplo, na Figura 6 são ilustrados os passos para inferência da expressão λ`f.`λ`x.f x` utilizando o Algoritmo W. O algoritmo é executado recursivamente para cada expressão, até que se tenha apenas a variável. Na figura 6, são ilustradas as chamadas recursivas e o retorno em cada passo está à direita do símbolo $\Rightarrow$. Em todos os passos é retornada uma dupla, composta por um conjunto de substituições e o tipo. Portanto, o tipo final retornado para a expressão λ`f.`λ`x.f x` é `(b` $\to$ `c)` $\to$ `b` $\to$ `c`.

O sistema de tipos de Damas-Milner descreve uma forma de inferência simples e expressiva, combinando flexibilidade (com possibilidade de definir expressões que implementam o conceito de polimorfismo paramétrico), robustez (devido a semântica consistente), além de permitir detectar erros em tempo de compilação (DAMAS; MILNER, 1982). O algo-

$$\frac{\Gamma(\mathtt{x}) = \forall\overline{\alpha}.\tau \qquad \mathtt{S} = [\overline{\alpha \mapsto \beta}] \qquad \beta \ \mathtt{fresh}}{\Gamma \vdash \mathtt{x} : (\mathtt{S}\tau, \mathtt{S})} \quad \text{(VAR)}$$

$$\frac{\begin{array}{ll}\Gamma \vdash \mathtt{e}_1 : (\tau_1, \mathtt{S}_1) & \Gamma \vdash \mathtt{e}_2 : (\tau_2, \mathtt{S}_2) \\ \mathtt{S'} = \mathtt{unify}(\tau_2 \to \beta \sim \mathtt{S}_2\tau_1) & \beta \ \mathtt{fresh} \\ \mathtt{S} = \mathtt{S'} \circ \mathtt{S}_1 \circ \mathtt{S}_2 & \end{array}}{\Gamma \vdash \mathtt{e}_1\ \mathtt{e}_2 : (\mathtt{S}\beta, \mathtt{S})} \quad \text{(APP)}$$

$$\frac{\Gamma, \mathtt{x} : \beta \vdash \mathtt{e} : (\tau, \mathtt{S}) \qquad \beta \ \mathtt{fresh}}{\Gamma \vdash \lambda\mathtt{x}.\mathtt{e} : (\mathtt{S}(\beta \to \tau), \mathtt{S})} \quad \text{(ABS)}$$

$$\frac{\begin{array}{ll}\Gamma, \mathtt{x} : \tau \vdash \mathtt{e} : (\tau', \mathtt{S}_1) & \Gamma, \mathtt{x} : \sigma \vdash \mathtt{e}' : (\tau'', \mathtt{S}_2) \\ \sigma = \mathtt{close}(\mathtt{S}_1\Gamma, \tau) & \mathtt{S} = \mathtt{S}_1 \circ \mathtt{S}_2 \end{array}}{\Gamma \vdash \mathtt{let}\ \mathtt{x} = \mathtt{e}\ \mathtt{in}\ \mathtt{e}' : (\tau'', \mathtt{S})} \quad \text{(LET)}$$

Figura 5: Algoritmo W: representação sob a forma de regras de tipo

ritmo é ao mesmo tempo consistente e completo[3] sem a necessidade de fornecer qualquer informação sobre tipo. No entanto, os sistemas de tipo têm evoluído muito, com o objetivo de tornar linguagens estaticamente tipadas mais expressivas. Com isso, várias novas características têm sido adicionadas, como é o caso do suporte a sobrecarga e tipos de dados algébricos generalizados, sendo necessário estender esse sistema.

2.3 Recursão Polimórfica

No sistema de Damas-Milner, toda chamada recursiva de uma função em sua definição deve ter o mesmo tipo. Recursão polimórfica ocorre quando uma função é usada com mais de um tipo na expressão

[3] Consistente e completo em relação ao conjunto de regras que definem o sistema de tipos. Consistente uma vez que todas as suposições de tipo inferidas pelo algoritmo podem ser provadas. E completo porque toda suposição que pode ser provada, também pode ser inferida pelo algoritmo.

$$
\begin{array}{lll}
W(\texttt{[ ]},\lambda\texttt{f.}(\lambda\texttt{x.f x})) = & (\texttt{S}(\beta \to \tau)\texttt{, S}) = & W(\texttt{[f:a]},\lambda\texttt{x.(f x)}) \\
 & & \texttt{onde: } \beta \mapsto \texttt{a} \\
 & \Rightarrow \texttt{((b} \to \texttt{c)} \to \texttt{b} \to \texttt{c, a} \mapsto \texttt{b} \to \texttt{c)} & \\
 & & \\
W(\texttt{[f:a]},\lambda\texttt{x.(f x)}) = & (\texttt{S}(\beta \to \tau)\texttt{, S}) = & W(\texttt{[f:a,x:b],f(x)}) \\
 & & \texttt{onde: } \beta \mapsto \texttt{b} \\
 & \Rightarrow \texttt{(b} \to \texttt{c, a} \mapsto \texttt{b} \to \texttt{c)} & \\
 & & \\
W(\texttt{[f:a,x:b],(f x)}) = & (\texttt{S}\beta\texttt{, S} \circ \texttt{S}_2 \circ \texttt{S}_1\texttt{)} = & W(\texttt{[f:a,x:b],f}) \\
 & & W(\texttt{[f:a,x:b],x}) \\
 & & \texttt{onde: } \beta \mapsto \texttt{c} \\
 & \Rightarrow \texttt{(c, a} \mapsto \texttt{b} \to \texttt{c)} & \\
 & & \\
W(\texttt{[f:a,x:b],f}) = & \texttt{(a, Id)} & \\
 & & \\
W(\texttt{[f:a,x:b],f}) = & \texttt{(b, Id)} &
\end{array}
$$

Figura 6: Passos para inferência da expressão $\lambda\texttt{f.}(\lambda\texttt{x.f x})$ usando o Algoritmo W

de definição da própria função, geralmente devido a uso de tipos não uniformes. A Figura 7 ilustra a regra que estende o sistema de tipos de Damas-Milner adicionando suporte a recursão polimórfica. Esse sistema é conhecido como sistema de Milner-Mycroft (MYCROFT, 1984). A regra LETREC-P é semelhante a regra LETREC-M apresentada na Seção 2.2, a diferença é que `x` tem tipo quantificado tanto em `e` quanto em `e'`, ou seja, `x` é polimórfico inclusive no contexto em que o tipo de `e` é inferido.

É possível entender a aplicabilidade deste recurso a partir do exemplo a seguir, onde o tipo `Seq t` é proposto por (OKASAKI, 1998) para representação de árvores binárias perfeitamente balanceadas:

```
data Seq t = Nil | Cons t (Seq (t,t))
```

O tipo `Seq` não é uniforme, devido ao fato de que o componente

$$\frac{\Gamma, \texttt{x} : \sigma \vdash \texttt{e} : \tau \quad \Gamma, \texttt{x} : \sigma \vdash \texttt{e'} : \tau\text{'} \quad \sigma = \texttt{close}(\Gamma, \tau)}{\Gamma \vdash (\texttt{let x } = \texttt{e in e'}) : \tau\text{'}} \quad \text{(LETREC-P)}$$

Figura 7: Regra que estende o sistema de Damas-Milner para suporte à recursão polimórfica

recursivo `Seq (t,t)` é estruturalmente diferente do tipo `Seq t`. Algoritmos envolvendo tipos não uniformes geralmente são mais eficientes, se comparados com a implementação uniforme equivalente. Como exemplo, a função `length` a seguir, que calcula o tamanho de uma sequência:

```
length Nil = 0
length (Cons x s) = 1 + 2 * (length s)
```

Este algoritmo executa em tempo $O(\log n)$, enquanto que a função `length` sobre listas executa em tempo $O(n)$. Entretanto, essa definição da função `length` não seria aceita em uma linguagem de programação baseada no sistema de tipos de Damas-Milner, uma vez que a chamada recursiva de `length`, nesta definição, não tem o mesmo tipo monomórfico da função que está sendo definida. Esta definição seria aceita apenas com o uso de extensões e mediante o uso de assinatura de tipos.

Tipos não uniformes podem ser convertidos para uma versão uniforme, porém é necessário adicionar um novo tipo que representa o componente não uniforme. No caso de `Seq`, na versão uniforme criamos um tipo chamado `Elem` que representa os elementos da sequencia:

```
data Elem t = Elem t | Pair (Elem t) (Elem t)
data Seq t  = Nil | Cons (Elem t) (Seq t)
```

Uma consequência do uso da versão uniforme é o aumento do número de construtores, o que pode diminuir a eficiência do algoritmo, devido ao aumento da estrutura que se precisa percorrer. A função `length`, reescrita para utilizar a versão uniforme, teria a seguinte forma:

```
lengthS Nil = 0
lengthS (Cons e t2) = 1 + lengthE e + lengthS t2

lengthE (Elem t) = 1
lengthE (Pair t1 t2) = lengthE t1 + lengthE t2
```

A função `lengthS` percorre o tipo de dado `Seq` somando 1 sempre que o construtor `Cons` for utilizado, já a função `lengthE` percorre o tipo de dado `Elem` e adiciona 1 sempre que o construtor `Elem`[4] for utilizado.

Além de questões relacionadas a eficiência do algoritmo, a versão uniforme não garante a construção de uma árvore perfeitamente balanceada, pois torna mais flexível a criação de um elemento (que pode ou não ser um par). Resumidamente, tem-se que a versão não uniforme tem algumas vantagens com relação à versão uniforme, que são[5]:

- Definição do tipo de dado é mais concisa;
- Menos construtores são necessários para representar o tipo de dado, o que pode torná-lo mais eficiente (algoritmos que utilizam a versão não uniforme precisam percorrer apenas os construtores `Cons` e `Nil`, enquanto que a versão uniforme cria novos construtores — `Pair` e `Elem`);
- Tipos não uniformes facilitam a percepção do tipo que está sendo gerado, pois é explícito que o primeiro elemento da sequência (do

[4] Neste exemplo foi utilizado o mesmo nome (`Elem`) para representar o construtor de tipo e o construtor do dado, permitido pelo compilador Haskell GHC.

[5] Em (OKASAKI, 1998) há uma discussão mais detalhada sobre as razões para se preferir a definição não uniforme de `Seq t` em relação à definição uniforme

tipo `Seq` acima) é simples, o segundo é um par, o terceiro um par de pares, e assim sucessivamente;

- A versão não uniforme garante que será construída uma árvore perfeitamente balanceada, o que não ocorre com a versão não uniforme, que permite construções como:
  `Cons (Pair (Elem 2) (Pair (Elem 3) (Elem 4))) Nil` — Apenas um elemento à esquerda
  `Cons (Pair (Elem 2) (Elem 3)) (Cons (Elem 5) Nil)` — Um par a esquerda, e à direita uma nova sequência.

Haskell, assim como ML, não permite que uma função que está sendo definida seja usada polimorficamente no contexto de sua própria definição, a menos que seja informado o tipo polimórfico explicitamente na definição da função, como ocorre com a função `lenght`, que pode ser assinada com tipo `length :: Seq a` $\rightarrow$ `Int`. A inferência de tipos em um sistema de tipos com suporte para recursão polimórfica pode ser reduzida ao problema da semi-unificação (HENGLEIN, 1993). Kfoury et al. (KFOURY; TIURYN; URZYCZYN, 1993) provaram que o problema da semi-unificação é indecidível.

O problema da semi-unificação é uma generalização do problema da unificação (JAHAMA; KFOURY, 1993). Dado um conjunto de pares de tipos $\{(\tau_i, \tau_i')\}^{i=1\ldots n}$, este problema consiste em decidir se existe uma substituição $\mathtt{S}$ e um conjunto de substituições $\{\mathtt{S}_1, \mathtt{S}_2, \ldots, \mathtt{S}_n\}$ tais que:

$$\mathtt{S}_1\mathtt{S}\tau_1 = \mathtt{S}\tau_1\text{'},\ \mathtt{S}_2\mathtt{S}\tau_2 = \mathtt{S}\tau_2\text{'}, \ldots,\ \mathtt{S}_n\mathtt{S}\tau_n = \mathtt{S}\tau_n\text{'}$$

Normalmente, uma instância deste problema é representada na forma de um conjunto de inequações com índices:

$$\{\tau_1 \leq^1 \tau_1', \tau_2 \leq^2 \tau_2', \ldots \tau_n \leq^n \tau_n'\}$$

Na definição de `Seq`, por exemplo, este problema seria representado como $\{$`[Seq` $a_1 \mapsto$ `Int]` $\leq$ `[Seq` $(a_2, a_2) \mapsto$ `Int]`$\}$ e pode ser resolvido pelas seguintes substituições: $S_1 = [a_1 \mapsto (a_2, a_2)]$ e `S = [id]`.

Recursão polimórfica pode ocorrer em funções que manipulam GADTs, que manipulam ADTs (com menor frequência), e também na definição de funções sobrecarregas, como será visto a seguir.

2.4 Sobrecarga

O sistema de tipos de Damas-Milner, possibilita definir expressões que operam sob diferentes tipos, desde que o comportamento seja uniforme, característica que comumente chamamos de *polimorfismo paramétrico* (ou *polimorfismo-via-let*).

Apesar de bastante expressivo, no sistema de tipos de Damas-Milner a definição de símbolos cujo comportamento seja distinto para diferentes tipos (conhecido como *polimorfismo de sobrecarga* ou *polimorfismo ad-hoc*) não é suportada. Dois exemplos que demonstram a utilidade deste tipo de polimorfismo são a implementação do operador de adição `(+)`, que é sobrecarregado em muitas linguagens para operar com valores inteiros ou ponto flutuante, e o símbolo de igualdade `(==)`, sobrecarregado para operar com caracteres, inteiros, booleanos, entre outros.

Haskell estende o sistema de tipos de Damas-Milner, adicionando classes de tipos para suporte a definição de símbolos sobrecarregados. Diferentemente de outras linguagens de programação, o tratamento de sobrecarga em Haskell é uniforme, independente do símbolo.

Uma classe em Haskell pode ser vista como um agrupador de símbolos que serão sobrecarregados para diferentes tipos, especificando uma assinatura para cada símbolo (ou nome), que define o seu tipo principal. Os símbolos sobrecarregados são implementados como instâncias dessa classe, cada qual com um tipo específico. Considere como exemplo a declaração da classe `Eq` e a definição de três instâncias, uma que implementa a comparação de igualdade para inteiros, outra para caracteres, e outra para listas:

```
class Eq a where
      (==) :: a → a → Bool
instance Eq Int where
      x == y = eqInt x y
instance Eq Char where
      x == y = eqChar x y
instance Eq a ⇒ Eq [a] where
      [ ] == [ ] = True
      (x:xs) == (y:ys) = x == y && xs == ys
```

Supondo que as funções `eqInt` e `eqChar` são funções primitivas com tipos `Int → Int → Bool` e `Char → Char → Bool`, respectivamente, temos que as expressões `1 + 1 == 2`, `'a' == 'b'` e `[1,2] == [3,4]` são bem tipadas. No entanto, expressões como `[1.2,1.5] == 3` serão rejeitadas. Na definição da instância de `Eq` para listas, uma restrição de tipo é imposta: `Eq a ⇒ Eq [a]`, esta restrição indica que apenas listas de tipos que são instâncias de `Eq` são aceitas, rejeitando expressões inválidas, como a igualdade de listas de funções, por exemplo.

Símbolos sobrecarregados podem ser usados na definição de funções polimórficas, tal como no exemplo a seguir, que apresenta a definição da função `ins`, que insere um novo elemento no final de uma lista dada, caso ele não ocorra nesta lista:

```
ins a [ ] = [a]
ins a (x:xs) = if a == x then x:xs
               else x:(ins a xs)
```

A função `ins` tem tipo $\forall$`a.{Eq a}.a` $\rightarrow$ `[a]` $\rightarrow$ `[a]`, sendo `Eq a` uma restrição de tipo, que indica que variável `a` pode ser instanciada apenas com tipos que são instâncias da classe `Eq`.

Classes de tipos podem definir uma hierarquia de classes, tal como ilustra a definição da classe `Ord` a seguir, a qual é definida como subclasse de `Eq`. Com esta definição, um tipo só poderá ser declarado como instância de `Ord`, se houver uma declaração deste tipo como instância de `Eq`:

```
class Eq a where
      (==) :: a → a → Bool

class (Eq a) ⇒ Ord a where
      (<) :: a → a → Bool
      (>) :: a → a → Bool
```

Neste caso, dizemos que `Eq` é uma superclasse da classe `Ord`. O uso de superclasse permite simplificar o tipo de expressões: supondo que a definição de uma função `f` utilize os símbolos (==) e (>), com o uso de superclasse o tipo inferido contempla restrição apenas para classe `Ord`; do contrário, as classes `Eq` e `Ord` são inclusas na restrição de `f`.

Classes de tipos permitem também a definição de funções *default*, tal como ilustrado no exemplo a seguir:

```
class Eq a where
      (==), (/=):: a → a → Bool
      x == y = not (x /= y)
      x /= y = not (x == y)
```

Com isso, apenas uma das funções, (==) ou (/=), precisa ser definida para um novo tipo, a implementação *default* será usada para a função que for omitida:

```
data Bool = True | False

instance Eq Bool where
         True  == True   = True
         False == False  = True
         _     /= _      = False
```

A definição de instâncias pode ser automatizada por meio de derivação automática de código, implementada para classes da biblioteca padrão de Haskell, como `Eq`, `Ord`, `Show`, `Enum` e `Read`. Utilizando a cláusula `deriving` associada à definição de novos tipos, o código referente à instância deste tipo de dado para a classe informada será gerado automaticamente, sem que o programador tenha que implementá-lo.

```
data Bool = True | False deriving (Eq,Show)
```

Haskell, com o uso de classes de tipos, adota uma abordagem de *mundo aberto* para sobrecarga. Na classe é definido o tipo principal para o símbolo, e toda possível instância da classe é considerada na solução da sobrecarga do símbolo num dado contexto. Em uma abordagem de *mundo fechado*, ao contrário, as instâncias do símbolo sobrecarregado que serão considerados são restringidas, seja por uma opção do programador (classes de tipo em Haskell permitem limitar as instâncias que podem ser criadas, restringindo o tipo principal na definição do símbolo na classe), seja pelo contexto de uso (como ocorre no Sistema CT, explicado na Seção 2.5), ou outros motivos que atribuam restrições na criação de instâncias para um símbolo.

Todos os exemplos apresentados até aqui ilustram o uso de símbolo sobrecarregado cujo tipo tem apenas um parâmetro, porém existem vários casos onde mais de um parâmetro de tipo é necessário. Como exemplo, considere uma operação de multiplicação envolvendo diferentes tipos, tal como abaixo. Considerando que o construtor `Vector` é definido como `Vector Int Int` e `Matrix` como `Matrix Vector Vector`, é possível obter diferentes combinações a serem tratadas para o operador `(*)`:

```
(*) :: Matrix → Matrix → Matrix
(*) :: Matrix → Vector → Vector
(*) :: Matrix → Int → Matrix
(*) :: Int → Matrix → Matrix
```

Para casos como esse, a classe de tipos teria que ser definida conforme abaixo:

```
class Mult a b c where
  (*) :: a → b → c
```

Essa declaração não é permitida pelo compilador GHC, devido à possibilidade de ocorrer ambiguidade, ou seja, uma entrada pode ser válida para mais de uma instância, e o compilador teria que "decidir" uma opção. Apesar de aumentar muito a flexibilidade no uso de símbolos sobrecarregados, classes de tipos com múltiplos parâmetros não faz parte da padronização da linguagem, mas existem extensões que permitem o uso, por exemplo, a partir de dependência funcional (JONES, 2000).

2.5 Sistema CT

O sistema CT é uma extensão do sistema Damas-Milner que adota uma abordagem de mundo fechado para o tratamento de sobrecarga. O tipo principal de um símbolo sobrecarregado é definido por meio de anti-unificação sobre os tipos das definições desse símbolo. O tipo principal de uma expressão é representado na forma $\forall \mathtt{a}_1 ... \forall \mathtt{a}_n.\kappa.\tau$, sendo que a restrição κ é um conjunto, possivelmente vazio de restrições de tipo, na forma de pares $o : \tau$, sendo o um símbolo sobrecarregado e τ um tipo simples.

O processo de anti-unificação originalmente proposto por Plotkin (1970), é usado para generalização de tipos. Dizemos que um tipo τ_g é uma generalização dos tipos τ_1 e τ_2 se existem substituições $\mathtt{S}_1$ e $\mathtt{S}_2$ tais que $\mathtt{S}_1(\tau_g) = \tau_1$ e $\mathtt{S}_2(\tau_g) = \tau_2$. O processo de obter a generalização mais específica é chamado de anti-unificação.

Chamamos a função que descreve a anti-unificação de um conjunto de tipos $\overline{\tau} = \{\tau_1, \ldots, \tau_n\}$ de *Least Common Generalization* e apresentamos um algoritmo para implemenção dessa função na Figura 8. A implementação usa mapeamentos $\mathtt{S}$ de variáveis de tipo (α) em pares de tipos (τ_1, τ_2) de modo a "lembrar" generalizações já realizadas. Por exemplo, `lcg` aplicada ao conjunto de tipos $\{\alpha_1 \rightarrow \beta_1 \rightarrow \alpha_1, \alpha_2 \rightarrow \beta_2 \rightarrow \alpha_2\}$ `retorna` $\alpha \rightarrow \beta \rightarrow \alpha$, onde α e β são novas variáveis de tipo, e não, digamos, $\alpha \rightarrow \beta \rightarrow \alpha'$.

Para ilustrar o funcionando do sistema CT no tratamento de sobrecarga, considere um contexto onde o operador de igualdade tem definições sobrecarregadas para os tipos `Int` e `Char`:

$$\Gamma_{(==)} \ \{(==) : \mathtt{Int} \rightarrow \mathtt{Int} \rightarrow \mathtt{Bool},\ (==) : \mathtt{Char} \rightarrow \mathtt{Char} \rightarrow \mathtt{Bool}\}$$

o tipo principal inferido para `(==)` no sistema CT é:

$$\forall \mathtt{a}.\{(==) : \mathtt{a} \rightarrow \mathtt{a} \rightarrow \mathtt{Bool}\}.\mathtt{a} \rightarrow \mathtt{a} \rightarrow \mathtt{Bool}$$

$$\texttt{lcg}(\mathbb{T}) = \tau \quad \texttt{where } (\tau,\texttt{S}) = \texttt{lcg}'(\mathbb{T},\emptyset)\texttt{, for some S}$$

$$\texttt{lcg}'(\{\tau\},\texttt{S}) = (\tau,\texttt{S})$$

$$\texttt{lcg}'(\{\tau_1,\tau_2\}\cup\mathbb{T},\texttt{S}) = \texttt{lcg"}(\tau,\tau',\texttt{S}') \quad \texttt{where} \quad \begin{array}{ll}(\tau,\texttt{S}_0) & = \texttt{lcg}''(\tau_1,\tau_2,\texttt{S})\\ (\tau',\texttt{S}') & = \texttt{lcg}'(\mathbb{T},\texttt{S}_0)\end{array}$$

$$\begin{array}{l}
\texttt{lcg}''(\texttt{C}\,\overline{\tau}^{\,n},\ \texttt{D}\,\overline{\rho}^{\,m},\texttt{S}) = \\
\quad \textbf{if } \texttt{S}(\alpha) = (\texttt{C}\,\overline{\tau}^{\,n},\ \texttt{D}\,\overline{\rho}^{\,m}) \texttt{ for some } \alpha \textbf{ then } (\alpha,\texttt{S})\\
\quad \textbf{else}\\
\qquad \textbf{if } \texttt{n} \neq \texttt{m} \textbf{ then } (\beta,\texttt{S}[\beta\mapsto(\texttt{C}\,\overline{\tau}^{\,n},\ \texttt{D}\,\overline{\rho}^{\,m})])\\
\qquad\quad \texttt{where } \beta \texttt{ is a fresh type variable}\\
\qquad \textbf{else } (\psi\,\overline{\tau'}^{\,n},\texttt{S}_n)\\
\qquad\quad \texttt{where} \quad (\psi,\texttt{S}_0) = \begin{cases}(\texttt{C},\texttt{S}) & \texttt{if C} = \texttt{D}\\ (\alpha,\texttt{S}\,[\alpha\mapsto(\texttt{C},\texttt{D})]) & \texttt{otherwise } \alpha \texttt{ is fresh}\end{cases}\\
\qquad\qquad\qquad (\tau_i',\texttt{S}_i) = \texttt{lcg}''(\tau_i,\rho_i,\texttt{S}_{i-1}),\ \texttt{for i} = 1,\ldots,\texttt{n}
\end{array}$$

Figura 8: Anti-Unificação

obtido por meio da anti-unificação (`lcg`) das definições sobrecarregadas no contexto.

O uso de um símbolo sobrecarregado em uma expressão faz com que uma restrição seja incluída no tipo inferido. Por exemplo, considere a seguinte definição da função `insert`, para inserir um elemento em uma lista:

```
insert x [ ]      = [x]
insert x (y:ys)   = if x == y then y:ys else y:(insert x ys)
```

O tipo inferido para essa função, no contexto $\Gamma_{(==)}$ é:

$$\forall \texttt{a}.\{(\texttt{==}) : \texttt{a} \rightarrow \texttt{a} \rightarrow \texttt{Bool}\}.\texttt{a} \rightarrow \texttt{[a]} \rightarrow \texttt{[a]}$$

Neste tipo, a restrição $\{(\texttt{==}) : \texttt{a} \rightarrow \texttt{a} \rightarrow \texttt{Bool}\}$ significa que a função `insert` somente pode ser aplicada a listas de elementos com um tipo τ para o qual o operador `(==)` esteja definido, isto é, um tipo τ tal que $(\texttt{==}) : \tau \rightarrow \tau \rightarrow \texttt{Bool} \in \Gamma_{(\texttt{==})}$.

De modo geral, o uso de uma expressão e com tipo $\sigma = \forall \texttt{a}_1 \ldots \texttt{a}_n.\kappa.\tau$ restringe o conjunto de tipos para os quais o tipo de `e` pode ser instanciado em um contexto de tipos Γ, de acordo com suposições de tipo de Γ que *satisfazem* as restrições que ocorrem em κ.

O problema da satisfazibilidade de um conjunto de restrições consiste em um problema para determinar, se para um par (Γ, κ), onde Γ é um contexto de tipos e κ é um conjunto de restrições, $\Gamma \models_{\texttt{S}} \kappa$ é provável, para alguma substituição `S`, que é então chamada de solução do problema de satisfazibilidade. A definição de CS-SAT é dada a seguir.

$$\frac{}{\Gamma \models_{id} \emptyset} \qquad (\text{SAT}_0)$$

$$\frac{\texttt{S}\tau = \texttt{S}\tau\texttt{'} \qquad \Gamma \models_{\texttt{S'}} \texttt{S}\kappa}{\Gamma \cup \{\texttt{x} : \forall\overline{\alpha}.\kappa.\tau\texttt{'} \models_{\texttt{S'}\circ\texttt{S}} \{\texttt{x} : \tau\}\}} \qquad (\text{SAT}_1)$$

$$\frac{\Gamma \models_{\texttt{S}} \{\texttt{x} : \tau\} \qquad \Gamma \models_{\texttt{S}} \kappa}{\Gamma \models_{\texttt{S}} \kappa \cup \{\texttt{x} : \tau\}} \qquad (\text{SAT}_n)$$

A solução `S` é dita principal se para qualquer outra solução `S'` existe uma substituição `R` tal que `S' = R` $\circ$ `S`. A aplicação da solução principal é chamado aprimoramento (CAMARÃO et al., 2007).

Um algoritmo para resolver CS-SAT é descrito como uma função denominada `sat` em (CAMARAO; FIGUEIREDO; VASCONCELLOS, 2004). Nos casos em que as restrições são satisfeitas o retorno desse algoritmo é uma substituição que pode ser usada no aprimorando do tipo. Essa substituição também é empregada na inferência de funções sobrecarregadas em que ocorrem recursão polimórfica. Um exemplo é a inferência de uma nova sobrecarga para o símbolo (==) no contexto $\Gamma_{(==)}$:

```
[ ] == [ ] = True
(x:xs) == (y:ys) = x == y && xs == ys
```

Nesse exemplo, não é possível determinar o tipo principal do símbolo sobrecarregado (==) aplicado de forma recursiva, uma vez que a nova definição sobrecarregada está sendo adicionada ao contexto e essa definição apresenta uma chamada recursiva com um tipo distinto. Nessa situação, a restrição é gerada considerando os tipos requeridos pelas chamadas recursivas, tal como apresentado abaixo, e não é possível determinar, neste momento, que os dois parâmetros para o operador (==) devem ter o mesmo tipo:

$$\{\texttt{(==): a} \rightarrow \texttt{b} \rightarrow \texttt{Bool,}$$
$$\texttt{(==): [a]} \rightarrow \texttt{[b]} \rightarrow \texttt{Bool}\}\texttt{.[a]} \rightarrow \texttt{[b]} \rightarrow \texttt{Bool}$$

CS-SAT retorna o seguinte conjunto de substituições que satisfazem as restrições:

$$\{\{\texttt{a} \mapsto \texttt{Int, b} \mapsto \texttt{Int}\}, \{\texttt{a} \mapsto \texttt{Char, b} \mapsto \texttt{Char}\},$$
$$\{\texttt{a} \mapsto \texttt{[Int], b} \mapsto \texttt{[Int]}\}, \{\texttt{a} \mapsto \texttt{[Char], b} \mapsto \texttt{[Char]}\},$$
$$\{\texttt{a} \mapsto \texttt{[[Int]], b} \mapsto \texttt{[[Int]]}\}, \{\texttt{a} \mapsto \texttt{[[Char]], b} \mapsto \texttt{[[Char]]}\},$$
$$\ldots\}$$

A generalização dessas substituições, obtida por meio da anti-unificação, é usada para o refinamento do tipo, depois de aprimorado e simplificado obtemos o tipo principal:

$$\forall \mathtt{a}.\{\mathtt{(==)}: \mathtt{a} \rightarrow \mathtt{a} \rightarrow \mathtt{Bool}\}.\mathtt{[a]} \rightarrow \mathtt{[a]} \rightarrow \mathtt{Bool}$$

Na abordagem proposta nesse trabalho a função `sat` é utilizada para inferir o tipo de funções onde ocorrem GADTs e que apresentam recursão polimórfica. Embora CS-SAT seja um problema indecidível é possível limitar o conjunto de restrições aceitas de forma a garantir a terminação da verificação. A terminação do algoritmo para CS-SAT é tratada em (RIBEIRO; CAMARAO; FIGUEIREDO, 2013).

3 TIPOS DE DADOS GENERALIZADOS

Este capítulo apresenta uma breve revisão sobre tipos de dados algébricos (ADTs) e tipos de dados algébricos generalizados (GADTs) e discute as principais abordagens para inferência de tipos de funções que utilizam GADTs.

3.1 Tipos Algébricos de Dados

Haskell possibilita a definição de novos tipos pelo programador, denominados tipos algébricos de dados (ADTs), os quais são definidos por meio da cláusula *data*. Os exemplos a seguir ilustram definições de tipos de dados algébricos:

```
data Bool = True | False

data Tree a = No a (Tree a) (Tree a) | Leaf
```

Uma definição de tipo de dado algébrico introduz um novo tipo, juntamente com os construtores de valores desse tipo. Por exemplo, o tipo `Bool`, definido acima, tem dois construtores – `True` e `False`. Um tipo de dado algébrico pode ser parametrizado por variáveis de tipo, como no caso da definição do tipo `Tree a` acima, onde a variável de tipo `a` representa o tipo dos valores contidos na árvore. O tipo `Tree a` é um tipo recursivo e possui dois construtores: `Leaf`, que representa as folhas da árvores e `No`, que representa os nodos internos da árvore, com subárvores à esquerda e à direita, do mesmo tipo `Tree a`. O argumento de tipo pode ser instanciado para diferentes tipos: inteiros, caracteres, strings, booleanos, ou novos ADTs definidos pelo programador, desde que este tipo seja *uniforme* para todos os elementos da árvore.

Os tipos dos construtores da árvore são inferidos pelo compilador como sendo:

```
No :: a → Tree a → Tree a → Tree a
Leaf :: Tree a
```

A definição de funções que operam sobre um ADT geralmente utiliza de casamento de padrão para desconstrução do ADT, onde cada alternativa corresponde a um construtor de dado desse tipo e especifica a expressão que será executada para essa alternativa, tal como ilustra a função `lenghtTree` definida a seguir:

```
lenghtTree :: Tree a → Int
lenghtTree Leaf = 0
lenghtTree (No a l r) = 1 + lenghtTree l + lenghtTree r
```

A função `lenghtTree` extrai o número de elementos armazenados em uma árvore, sendo que cada alternativa da definição desta função corresponde a um construtor do tipo `Tree a`.

3.2 Tipos de Dados Algébricos Generalizados

Tipos de dados algébricos generalizados (GADTs) constituem uma poderosa generalização de tipos algébricos de dados em Haskell e ML[1]. GADTs são uma extensão dos ADTs com suporte a construtores de dado com tipos *não uniformes*. Peyton Jones et al. (JONES; WASHBURN; WEIRICH, 2004; JONES et al., 2006) foram os primeiros a usar o termo *Generalized Algebraic Data Types* para se referir a esta extensão dos ADTs, e definiram a sintaxe para declaração deste tipo de dado no compilador GHC.

[1] GADTs também são encontrados na literatura como “guarded recursive data types” (XI; CHEN; CHEN, 2003) ou “first-class phantom types” (CHENEY; HINZE, 2003).

O compilador GHC suporta GADTs desde a versão 6.4 em 2005 (TEAM, 2005) §7.5, com base em *Wobbly Types* (JONES; WASHBURN; WEIRICH, 2004), onde a assinatura de tipo era obrigatória em toda expressão que utilizasse GADTs. Versões mais recentes do GHC baseiam-se nos trabalhos (JONES et al., 2006; VYTINIOTIS et al., 2011), onde é possível inferir o tipo de um conjunto restrito de expressões. Várias abordagens têm sido propostas com o objetivo de expandir o conjunto de programas aceitos como corretos, como será discutido na Seção 3.4.

Um GADT é definido listando os seus construtores e provendo uma assinatura explícita de tipo para cada construtor, o que permite a definição de funções que retornam expressões com tipos distintos, todos instâncias do tipo GADT. A definição do tipo de dado `Term` e da função `eval` (apresentado no Capítulo 1), é um exemplo clássico que ilustra a motivação para o uso de GADTs[2].

Neste exemplo, os construtores `Lit`, `Inc`, `IsZ`, `If` e `Pair` tem tipos diferentes:

```
data Term a where
     Lit    :: Int → Term Int
     Inc    :: Term Int → Term Int
     IsZ    :: Term Int → Term Bool
     If     :: Term Bool → Term a → Term a → Term a
     Pair   :: Term a → Term b → Term (a,b)

eval :: Term a → a
eval (Lit i)      = i
eval (Inc t)      = 1 + eval t
eval (IsZ i)      = 0 == eval i
eval (If b c d)   = if eval b then eval c
                    else eval d
eval (Pair a b)   = (eval a, eval b)
```

2 Outras aplicações interessantes de GADTs são discutidas, por exemplo, em (CHENEY; HINZE, 2003; XI; CHEN; CHEN, 2003)

A desconstrução de um GADT, assim como num ADT, ocorre pelo uso de casamento de padrões, como na definição de `eval`. Porém, ao contrário dos ADTs, onde o tipo do resultado de cada alternativa é unificado, em funções definidas sobre GADTs o tipo do retorno pode variar: veja que a alternativa para o construtor `Lit` retorna `Term Int` e para o construtor `IsZ` retorna `Term Bool`. A definição da função `eval` usa refinamento de tipo em cada alternativa da definição, para evitar o encapsulamento do resultado em outro tipo de dado (que seria necessário caso `Term` fosse descrito utilizando apenas ADTs). Com isso, erros que poderiam ocorrer em tempo de execução são evitados.

Nem toda variação de tipos é aceitável no sistema de tipos GADT: *o retorno das alternativas pode variar apenas de acordo com o tipo do parâmetro GADT*. Esta restrição representa um requisito de coerência básico entre o padrão GADT da alternativa e o retorno. A função `eval` obedece esta restrição, o tipo `eval :: Term a → a` indica que o retorno `a` está de acordo com o tipo do parâmetro `Term a`. Por outro lado, a definição da função `wEval :: Term a → b`, definida a seguir, deve ser rejeitada pelo sistema de tipos.

```
wEval :: Term a → b
wEval (Lit i)= True -- Tipo Argumento GADT: Int,
                       Tipo Retorno: Bool
wEval (IsZ i)= 0    -- Tipo Argumento GADT: Bool,
                       Tipo Retorno: Int
```

Tipos que não estão associados ao GADT devem ser unificados, como ocorre normalmente e, portanto, a função `cEval` abaixo deve ser aceita pelo sistema de tipos:

```
cEval :: Term a → Int -- O tipo Int é obtido por meio da
                         unificação das alternativas
cEval (Lit i) = i -- Tipo Argumento GADT: Int,
                     Tipo Retorno: Int
cEval (IsZ i) = i -- Tipo Argumento GADT: Bool,
                     Tipo Retorno: Int
```

$$\frac{\mathsf{C} : \sigma \quad \sigma \leq \tau}{\Gamma \vdash \mathsf{C} : \tau} \quad \text{(CONS)}$$

$$\frac{\Gamma \vdash \mathtt{p} : \tau \quad \Gamma \vdash \mathtt{e} : \tau'}{\Gamma \vdash \mathtt{p} \to \mathtt{e} : \tau \to \tau'} \quad \text{(ALT)}$$

$$\frac{\Gamma \vdash \mathsf{e} : \tau \qquad \Gamma \vdash \mathsf{p}_i \to \mathsf{e'}_i : \tau \to \tau' \qquad \text{para } i = 1..n}{\Gamma \vdash \text{case e of} \{\mathsf{p}_i \to \mathsf{e'}_i\}^{i=1..n} : \tau'} \quad \text{(CASE)}$$

Figura 9: Sistema de tipos ADT

Desde a concepção dos GADTs, questões sobre a inferência vêm sendo discutidas. Lin (LIN, 2010) cita que o primeiro trabalho a investigar uma extensão para os ADTs que permitisse construtores com tipos não uniformes foi de Augustsson e Petersson em 1994 (AUGUSTSSON; PETERSSON, 1994). Este problema foi melhor discutido oito anos mais tarde, em 2002 (BAARS; SWIERSTRA, 2002; CHENEY; HINZE, 2002). Apesar dos vinte anos de estudo, o problema de inferência de GADTs continua em aberto.

3.3 Sistema de Tipos para ADT e GADT

Para tratar tipos algébricos, o sistema de tipo apresentado na Figura 2 é estendido, incluindo regras para derivação de tipos adicionais, apresentadas na Figura 9. A regra CONS especifica a derivação de tipos para construtores de dados; a regra ALT especifica a derivação de tipos de cada alternativa de uma expressão; e a regra CASE especifica a derivação de tipos de expressões com alternativas case definidas sobre tipos de dados algébricos.

Regras específicas para alternativas `case` definidas sobre tipos de dados algébricos generalizados, usadas na abordagem proposta neste trabalho, são apresentadas no Capítulo 4.

3.4 Abordagens para inferência de tipos para GADTs

Wobbly types (JONES; WASHBURN; WEIRICH, 2004) foi a primeira abordagem de inferência de tipos para funções utilizando GADTs, onde o tipo das alternativas é verificado de acordo com a assinatura, que deve ser informada pelo programador. A principal inovação está nos tipos *woobly*, que representam os tipos que não sofrem refinamento de tipos (introduzido pelos GADTs) e, portanto, o algoritmo de inferência nunca olha dentro de tipos *woobly*.

Além de ser a primeira abordagem proposta, foi também a primeira utilizada pelo compilador GHC (na versão 6.4 em 2005 (TEAM, 2005)). A premissa no desenvolvimento era: adicionar GADT em Haskell sem a necessidade de grandes alterações no sistema de tipos, com um sistema que fosse fácil de usar e implementar em Haskell, adicionando este recurso de uma forma conservadora, ou seja, afetando o mínimo possível o sistema de tipos.

Porém essa abordagem trouxe alguns problemas na propagação do tipo da assinatura para os escopos internos. Esses problemas foram corrigidos no OutsideIn (VYTINIOTIS et al., 2011), que propôs trazer todas as restrições para o escopo mais externo e então resolvê-las, como será mostrado com detalhes na Seção 3.4.1.

A proposta apresentada por Simonet e Pottier (POTTIER; RÉGIS-GIANAS, 2006) é bastante semelhante à de *wobbly types*: em *Stratified type inference* é definido um algoritmo de inferência em dois passos, separando a tradicional inferência de tipos no estilo Damas-Milner da propagação de anotações explicitas de tipos. Essa divisão torna o mecanismo de propagação dos tipos anotados mais eficiente que no *wobbly types*.

Type Inference via Herbrand Constraint Abduction (SULZMANN; SCHRIJVERS; STUCKEY, 2008), assim como mais tarde foi feito no OutsideIn, a ideia consiste em gerar restrições de implicação de fora do programa e então resolver estas restrições por abdução de restrição Herbrand, um método para resolver a inferência na ausência de fatos (que neste caso são representados por restrições Herbrand). Ao contrário do OutsideIn, expressões onde não é possível obter o tipo principal nem sempre são rejeitadas pelo sistema de tipos. Ao invés disso, Sulzmann et al. apresentam exemplos de programas envolvendo GADTs com infinitas soluções, e que podem ser aceitas por descartar certos tipos não “intuitivos”.

Conforme observado em (LIN, 2010), a ideia proposta na abordagem de Sulzmann et al. é desconsiderar várias soluções não intuitivas (portanto podendo aceitar expressões onde não há tipo principal) e ainda tornar a inferência de tipos completa e decidível para o sistema de tipos GADT restrito a tipos intuitivos e sensíveis. O problema de perda do tipo principal em programas envolvendo GADTs e a opção por inferir tipo mesmo para expressões que não possuam tipo principal é discutida mais detalhadamente na Seção 4.2. Nas seções a seguir, é discutido mais detalhadamente os trabalhos considerados mais relevantes neste contexto, por tratarem de inferência de tipos para expressões envolvendo GADTs sem requerer assinatura de tipos.

3.4.1 OutsideIn

O algoritmo proposto por Schrijvers et al. é bastante conservador, como bem descrito pelos autores em (SCHRIJVERS et al., 2009) §4.4. Determinados critérios são adotados no algoritmo de inferência de tipos, de modo a garantir que apenas sejam inferidos tipos para expressões que possuam tipo principal. Entretanto, os critérios adotados são muito restritivos, resultando na rejeição de grande número de programas corretos, mesmo programas para os quais se poderia atribuir um único tipo.

Em vez de unificações o OutsideIn gera restrições de igualdade que serão resovidas apenas no escopo mais externo, gerando uma substituição que é aplicada à expressão. Essa abordagem resolve o problema da propagação da assinatura de tipos. Para as alternativas que envolvem GADTs são também geradas restrições de implicação para a verificação da consistência do tipo de cada alternativa, uma vez que podem instanciar tipos distintos. Portanto, expressões onde não é feito o casamento de padrões, como `mPair` abaixo (extraído de (SCHRIJVERS et al., 2009)), são resolvidas apenas por unificação:

```
data Pair a b where
  MkP :: a → b → Pair a b

mPair x = MkP x True
```

Neste caso, são geradas as seguintes restrições para o tipo da função *mPair*, onde β representa o tipo de `x` (lado esquerdo da função) que é igual ao tipo do primeiro argumento do construtor `MkP`, representado por γ_1; γ_2 representa o segundo argumento, que tem tipo `Bool`, e o tipo da expressão (lado direito) é representado por α[3].

$$\beta \sim \gamma_1 \wedge \gamma_2 \sim \texttt{Bool} \ \wedge \ \alpha \sim \textbf{\texttt{Pair}} \ \gamma_1 \ \gamma_2$$

O tipo de `mPair` é então inferido como: $\beta \rightarrow \texttt{Pair} \ \beta \ \texttt{Bool}$, sendo obtido por meio das seguintes substituições $\{\alpha \mapsto \texttt{Pair} \ \beta \ \texttt{Bool}, \ \gamma_2 \mapsto \texttt{Bool}, \ \gamma_1 \mapsto \beta\}$, que constituem a solução mais geral para o conjunto de restrições de igualdade acima.

Considere agora a função `test1` abaixo, cujo tipo é definido por casamento de padrões:

```
data T a where
  T1 :: Int → T Bool
  T2 :: T a

test1 (T1 n) = n > 0
```

Neste caso, são geradas as seguintes restrições de igualdade (fora do casamento de padrões) e de implicação (dentro do casamento de padrões), para o tipo da função `test1`, onde α_x define o tipo do padrão `case`[4], `T` α denota o tipo do valor construído por `(T1 n)` e β denota o tipo do resultado desta alternativa:

$$\alpha_x \sim \texttt{T} \ \alpha \ \wedge \ (\alpha \sim \texttt{Bool} \supset \beta \sim \texttt{Bool})$$

[3] Seguindo uma notação semelhante à apresentada em (SCHRIJVERS et al., 2009), temos que α, β e γ representam variáveis de unificação, `a`, `b` e `c` variáveis *skolem*, que serão unificadas, $\sim$ denota igualdade de tipos e $\supset$ denota implicação.

[4] Alternativas num casamento de padrões equivalem a uma expressão `case`, que tem a forma `case x of pattern → body`, portanto a função `test1` poderia ser reescrita como `test1 x = case x of (T1 n) → n > 0`, assim, o valor de `x` construído no primeiro padrão é `T a`.

É possível observar, que neste caso, não existe um único unificador mais geral: tanto $\{\beta \mapsto \texttt{Bool}\}$ como $\{\beta \mapsto \alpha\}$ seriam soluções possíveis, o que significa que `test1` não tem tipo principal: usando a substituição $[\beta \mapsto \texttt{Bool}]$, o tipo de `test1` seria $\forall\alpha.\texttt{T}\ \alpha \to \texttt{Bool}$; usando $[\beta \mapsto \alpha]$, o tipo seria $\forall\alpha.\texttt{T}\ \alpha \to \alpha$ e nenhum dos dois tipos é instância do outro. Na abordagem de OutsideIn, a definição de `test1` é considerada inválida. O critério utilizado no algoritmo para evitar a perda da propriedade de tipo principal é não tentar resolver as restrições de implicação, considerando intocáveis (*untouchables*) as variáveis dentro dessas restrições (tal como β, nesse exemplo).

Pelas regras do algoritmo que resolve as restrições, *apenas substituições geradas a partir de restrições de igualdade podem ser utilizadas para resolver as restrições de implicação*, portanto, se for incluída uma nova alternativa que permita especializar o tipo de β, a expressão é aceita. Considere a função `test2` abaixo:

```
test2 (T1 n) = n > 0
test2 T2     = True
```

O tipo dessa função tem as seguintes restrições:

$$\alpha_x \sim \texttt{T}\ \alpha \wedge (\alpha \sim \texttt{Bool} \supset \beta \sim \texttt{Bool}) \wedge \alpha_x \sim \texttt{T}\ \alpha' \wedge \beta \sim \texttt{Bool}$$

onde o tipo do padrão da nova alternativa (referente a `T2`) gera a restrição $\alpha_x \sim \texttt{T}\ \alpha'$ e o tipo do retorno gera uma nova restrição de igualdade $\beta \sim \texttt{Bool}$, fora da restrição de implicação, e que, portanto, pode ser resolvida por meio de unificação. A partir dessas restrições, são geradas as seguintes substituições: $\{\alpha_x \mapsto \texttt{T}\ \alpha,\ \ \alpha' \mapsto \alpha,\ \ \beta \mapsto \texttt{Bool}\}$, resultando no tipo:

$$\texttt{test2 :: T}\ \alpha \to \texttt{Bool}$$

No exemplo a seguir, `test3`, é adicionado um argumento não ligado ao GADT. A função `test3` não tem tipo principal, e portanto, não é aceito na abordagem OutsideIn. O tipo da função `test3` não é resolvido pois não é possível definir um tipo para β fora do GADT (como foi feito em `test2`) já que o tipo de `r` é desconhecido.

```
test3 (T1 n) _ = n > 0
test3 T2     r = r
```

Porém, uma vez que o tipo de `r` possa ser determinado, como em `test4` abaixo, é possível resolver o tipo de β e a função será aceita.

```
test4 (T1 n) _ = n > 0
test4 T2     r = not r
```

É importante perceber que tanto `test1` quanto `test3` poderiam ser resolvidos adicionando assinatura de tipos, `test1 :: T a` $\rightarrow$ `Bool` e `test3 :: T a` $\rightarrow$ `Bool` $\rightarrow$ `Bool`.

Como é possível perceber pelos exemplos, o nome *OutsideIn* reflete exatamente o comportamento do algoritmo, que propaga informação de quais tipos devem ser unificados do escopo local para o escopo mais externo de cada alternativa do casamento de padrão do GADT, por meio da geração de restrições de igualdade, concluindo a inferência de tipos por meio da resolução dessas restrições. Comparado com trabalhos anteriores, esta abordagem oferece um mecanismo mais natural de propagação de tipos anotados, e possibilita mensagens de erro mais representativas. Porém, a inferência de tipos ocorre para um conjunto restrito de declarações, pois são definidas variáveis *untouchables* para que sejam inferidos apenas os tipos de funções onde seja possível garantir a propriedade de tipo principal.

Ao contrário do *Wobbly Types* (JONES; WASHBURN; WEIRICH, 2004) e *Stratified type inference* (POTTIER; RÉGIS-GIANAS, 2006), possibilita a inferência de tipos para alguns programas, porém segundo Lin (LIN, 2010) é pouco representativo. O conjunto inferido em OutsideIn, torna obrigatória informação sobre o tipo fora do GADT, ou seja, com base em outras alternativas na mesma função, o que não é realidade em muitas funções, inclusive `eval`.

3.4.2 Algoritmo P e Pointwise

Lin (LIN, 2010) propõe o algoritmo P, capaz de inferir (sem a necessidade de assinatura) o tipo de funções em que ocorrem GADTs. Esse algoritmo infere o tipo para cada uma das alternativas da expressão `case` separadamente e verifica a necessidade de generalização. Um exemplo dessa necessidade é o caso em que os tipos de cada um dos resultados (corpo da alternativa) não podem ser unificados.

Cada alternativa $\mathtt{p}_i \rightarrow \mathtt{e}_i$ é composta de um padrão $\mathtt{p}_i$ e um corpo $\mathtt{e}_i$, a generalização[5] é realizada apenas sobre os padrões $\mathtt{p}_i$, o tipo resultante desse processo é usado no refinamento do tipo de cada uma das alternativas. Após o refinamento esses tipos são reconciliados, nesse processo, quando o tipo de uma subexpressão presente no corpo não combina com os tipos das outras alternativas esse tipo é substituído pelo tipo da subexpressão correspondente na generalização do padrão. Em (LIN; SHEARD, 2010; LIN, 2010) é apresentada uma versão desse algoritmo, denominada Pointwise, capaz de aceitar um conjunto mais abrangente de programas, mas que necessita assinatura de tipos.

3.4.3 Chore

Em Chore[6] (CHEN; ERWIG, 2016), são usados tipos “choice” para representar o tipo de cada alternativa numa expressão *case*. Como já está claro neste ponto do trabalho, no refinamento de tipos, cada alternativa pode ter um tipo diferente, tornando a etapa de reconciliação de funções GADT especialmente difícil, pois é comum haver mais de um tipo capaz de representar a mesma expressão. A forma como Chore trata expressões onde não é possível obter um tipo principal consiste em atribuir um conjunto de escolhas, e este tipo é propagado durante toda a inferência. O tipo das expressões só é *fechado* para facilitar a leitura do programador, mas em todas as etapas da inferência, são considerados os tipos choice.

Considere como exemplo a função `flop2` extraída de (VYTINIOTIS et al., 2011) e utilizada em (CHEN; ERWIG, 2016):

5 Que é uma simplificação da anti-unificação
6 Chore é um acrônimo para *choice reconciliation*.

```
data R a where
  RI :: Int → R Int
  RB :: Bool → R Bool
  RC :: Char → R Char

flop2 e = case e of
            RI x → x
            RB x → x
```

Esta função tem tipo `flop2:D⟨R Int, R Bool⟩ → D⟨Int,Bool⟩`, onde `D⟨...⟩` representa um tipo choice. Esta tipo representa que o tipo do argumento pode ser uma das alternativas `R Int` ou `R Bool` e o tipo do resultado é, respectivamente, `Int` ou `Bool`. Essa correlação é estabelecida por meio do uso do mesmo nome choice `D` no tipo do argumento e no tipo do resultado.

A expressão `flop2` tem um único conjunto de escolhas, porém, expressões com tipos aninhados, como `param1o`, terão mais de um conjunto de tipos choice, um para cada cada tipo *scrutinee*. Assim, o tipo de `param1o` é `A⟨G Int Int → Int, G Bool Bool → B⟨Int, Int⟩⟩`, onde `A` e `B` são tipos choice que representam os tipos de `e` e `b` respectivamente.

```
data G a b where
  G1 :: a → G Int a
  G2 :: a → G Bool a

param1o e = case e of
              G1 i → i + 3
              G2 b → case b of
                       True → 4
                       False → 7
```

No sistema de tipos de Chore, tipos choice são também chamados de tipos variáveis (*variational types*) e são representados por ϕ, e os demais tipos são chamados de simples (*plain types*). No refinamento de tipos, cada alternativa de uma expressão *case* tem tipo no formato $\tau \rightarrow \phi$, ou seja, o tipo do padrão é simples, enquanto que o tipo do corpo da expressão pode variar.

Expressões com chamadas recursivas de tipo GADT envolvem recursão polimórfica, e são aceitas por Chore mediante a assinatura de tipos. Chore utilizada da mesma estratégia de (JONES et al., 2006; SCHRIJVERS et al., 2009), onde são usadas duas regras para inferência de tipos de construções LET: a regra LET usual e a regra LETA, que requer assinatura de tipo para o símbolo que está sendo definido.

Além do tratamento de tipos variáveis, a regra CASE possui outra particularidade, que é o algoritmo que verifica a coerência de um conjunto de escolhas (*coherent*), este algoritmo evita que funções como `cross`, que poderia ter tipo `R a` $\rightarrow$ `b`, sejam aceitas:

```
cross (RI x) = even x
cross (RB x) = 1
```

A função `cross` tem tipo $\texttt{D}\langle\texttt{R Int, R Bool}\rangle \rightarrow \texttt{D}\langle\texttt{Bool, Int}\rangle$, como choices são conjuntos, a ordem dos elementos não é verificada, sendo portanto necessária esta etapa de verificação.

Além da regra CASE, as regras APP e MAIN possuem particularidades que devem ser apresentadas. Na aplicação de expressões, devido à propagação de tipos choice (que, diferente do algoritmo *W*, não são *fechados*), as regras se tornam mais complexas. Esta regra é responsável por capturar um conjunto maior de erros, se comparado com outras abordagens, o que é tido como uma das grandes vantagens do uso de Chore. Três casos são avaliados por um conjunto de regras para determinar se a aplicação de funções é bem tipada ou não, são elas:

1. O tipo do argumento casa com o argumento de tipo exatamente. Neste caso será retornado um tipo ϕ, e a função é considerada bem tipada.
2. Em algumas alternativas, o tipo do argumento casa com o argumento de tipo. Neste caso, o tipo do retorno será uma combinação de tipos choice que casam, extraídos de ϕ.

3. Em nenhuma alternativa o tipo do argumento casa com o argumento de tipo. Neste caso, a aplicação é considera mal tipada.

Com os tipos choice propagados, e os operadores utilizados para controlar a aplicação de funções, erros devido ao uso incorreto de uma função podem ser evitados. Por exemplo, a função `h2`, que normalmente é aceita, será rejeitada por Chore, visto que no domínio `D` do tipo de `flop2` não existe nenhuma alternativa que case com o tipo `R Char`:

```
h2 = flop2 (RC 'a')
```

A regra MAIN foi adicionada ao sistema de tipos com o objetivo de prover um tipo fechado para o programador, basicamente substituindo choices por variáveis de tipo, e executando o refinamento de tipos associados ao GADT. Diferente de outras abordagens, Chore gera o tipo final, num formato compreensível para o programador, ao final de todas as etapas de inferência, utilizando um algoritmo de reconciliação (*reconcile*). Com isso, o tipo gerado para `flop2` será $\texttt{R a} \rightarrow \texttt{a}$. Da mesma forma como na regra CASE, a função `cross` não terá o tipo fechado, na reconciliação, devido as validações de tipo.

Chore traz uma nova forma de inferência, cuja principal mudança está na forma como os tipos são descritos. O uso de choice, aliado a propagação de tipo, permite detectar um volume maior de erros, se comparado com outras abordagens. No entanto, a propagação de tipos torna a regra de aplicação mais complexa, necessitando que as restrições sejam propagadas por todo processo de inferência. É necessário uma verificação de coerência tanto na aplicação de funções quanto na reconciliação de tipos.

Além disso, Chore não trata expressões com recursão polimórfica, portanto não será possível inferir o tipo de funções como *eval*, comum quando se refere a expressões envolvendo GADTs. Expressões com cases aninhados envolvendo GADTs também não serão inferidos, com a justificativa de que a introdução de suposições de tipo locais para alternativas internas torna o tipo *scrutinee* muito geral, e faz com que se perca o benefício de capturar mais erros de tipo.

4 INFERÊNCIA DE TIPOS PROPOSTA

A inferência de tipos para GADTs, como visto nos capítulos anteriores, é bastante complexa, por envolver refinamento do tipo, recursão polimórfica e devido à perda do tipo principal em alguns casos. Neste capítulo é apresentada uma nova abordagem para inferência de tipos envolvendo GADTs, que utiliza anti-unificação para capturar a relação entre o tipo das alternativas, unificando variáveis de tipo que não estão relacionadas ao GADT, e casos que envolvem recursão polimórfica são tratadas de forma semelhante a sobrecarga no sistema CT. Durante a inferência de uma expressão `case`, são geradas restrições de tipo, estas restrições são utilizadas apenas no refinamento de tipos, e são resolvidas e eliminadas gerando um tipo “fechado” para a função. Na abordagem proposta são tratados apenas casos em que definições GADT ocorrem no escopo mais externo, como será discutido na Seção 5.8.

As definições `gtv`, `gtc` e `rtv` são utilizadas para representar variáveis e construtores ligados a GADTs. A notação $\texttt{gtv}(\sigma)$ representa o conjunto de variáveis de tipo que ocorrem em σ como parâmetros para um GADT. $\texttt{gtc}(\tau)$ denota o conjunto de construtores de tipos para GADTs que ocorrem em τ. E $\texttt{rtv}(\tau)$ representa o conjunto de variáveis de tipo que ocorrem como parâmetro de tipos de dados algébricos recursivos em τ[1].

Restrições são usadas para expressar a relação entre o tipo do retorno de uma função e seus parâmetros. Em expressões *case* o tipo do parâmetro tem um construtor GADT. Também é utilizada a notação a seguir para retornar o conjunto de restrições que contém tipos que mencionam construtores GADT:

$$\kappa^x = \{x : \tau \in \kappa \mid \text{gtc}(\tau) \neq \emptyset\}$$

[1] Em Chore e P, variáveis ligadas ao GADT são chamadas de variáveis de índice; são variáveis onde pode ocorrer o refinamento de tipos, que na abordagem proposta neste trabalho, refere-se as variáveis associadas ao GADT.

4.1 Inferência de Tipo

Na abordagem proposta, a inferência de funções utilizando GADT envolve generalizar os tipos das alternativas, para identificar quais os tipos estão associados ao GADT e que, portanto, podem ser instanciados como tipos distintos em cada alternativa. Para isso, primeiro é resolvida a recursão polimórfica, gerando uma restrição para cada chamada recursiva da função que está tendo seu tipo inferido. Essas restrições são verificadas e resolvidas pelo algoritmo CS-SAT, conforme proposto no sistema CT. A substituição resultante desse processo é usada no aprimoramento do tipo de cada alternativa, tratando, dessa forma, os casos que envolvem recursão polimórfica.

Um segundo passo é generalizar o tipo de cada alternativa, utilizando anti-unificação, o que possibilita capturar a relação entre os tipos das alternativas e definir quais variáveis de tipo que são parâmetros para os GADTs, essas variáveis são skolemizadas, permanecendo inalteradas no processo de unificação dos tipo das alternativas. Antes de expor a definição do algoritmo efetivamente, considere, como exemplo, a inferência de tipo para a função `eval` (que será melhor explicada na Seção 4.2):

```
data Term a where
  Lit  :: Int → Term Int
  Inc  :: Term Int → Term Int

eval e = case e of
           (Lit i)   → i
           (Inc t)   → eval t + 1
```

Apenas a alternativa referente a `Inc` introduz restrição, pois possui chamada recursiva, para `Lit` o tipo é direto; temos assim os tipos abaixo para as alternativas de `eval`:

```
(Lit i)      { }.eval :: Term Int → Int
(Inc t)      {eval :: Term Int → Int}.
              eval :: Term Int → Int
```

A solução para a restrição imposta pela chamada recursiva `eval t` é direta, pois o algoritmo de CS-SAT retorna uma substituição onde o tipo refinado será o mesmo já inferido para a alternativa.

Tendo o tipo aprimorado para as alternativas, é possível então generalizar e unificar esses tipos, atribuindo um tipo final para `eval`. Existem dois possíveis tipos para `eval` neste caso, $\texttt{Term Int} \to \texttt{Int}$ ou $\texttt{Term a} \to \texttt{a}$ (devido ao tipo do GADT, que é `Term a`). O primeiro tipo, no entanto, é considerado mais preciso e é o inferido pela abordagem proposta neste trabalho. Apesar da possibilidade de generalizar o tipo das alternativas, na abordagem proposta, o conjunto de substituições retornada por CS-SAT (como será explicado no decorrer deste capítulo), permite que tipos mais precisos sejam retornados.

A seguir é apresentada a formalização do algoritmo, bem como outros exemplos e uma breve discussão sobre os principais pontos envolvidos na inferência de tipos.

4.1.1 Definição do Algoritmo

Para simplificar, considera-se uma linguagem que é essencialmente uma extensão de ML com funções GADT — não incluímos inferência de tipos de expressões com símbolos sobrecarregados.

O algoritmo proposto é definido na forma de um sistema de provas dirigido por sintaxe, para a prova de julgamentos da forma $\Delta \mid \Gamma \vdash \texttt{e} : (\kappa.\tau, \texttt{S})$, onde Δ é um contexto de nomes de definições de funções recursivas que contém restrições que são usadas no processo de aprimoramento de tipos em alternativas case envolvendo GADTs, $\kappa.\,\tau$ é o tipo inferido para $\texttt{e}$ e $\texttt{S}$ é uma substituição (usada para instanciar variáveis de tipo para obter o tipo $\kappa.\,\tau$). A notação $\delta(\texttt{x}, \tau, \Delta)$ associa, com um símbolo $\texttt{x}$ e um tipo τ, um conjunto de restrições $\{\texttt{x} : \tau\}$, se $\texttt{x}$ é um símbolo definido recursivamente, do contrário é associado um conjunto de restrições vazio, ou seja, $\delta(\texttt{x}, \tau, \Delta) = \texttt{if}\ \texttt{x} \in \Delta\ \texttt{then}\ \{\texttt{x} : \tau\}\ \texttt{else}\ \emptyset$.

As regras para inferência de tipo são tal como no sistema de Damas Milner, com exceção das regras VAR e CASE. A regra VAR gera uma restrição para cada símbolo em Δ, usado para o refinamento de tipos de funções GADT. Cada variável $\texttt{x}$ que não está em Δ tem tipo com um conjunto de restrições vazio.

$$\frac{\mathtt{x} : \forall\overline{\alpha}.\tau \in \Gamma \quad \mathtt{S} = [\overline{\alpha \mapsto \beta}] \quad \overline{\beta}\ \mathit{fresh} \quad \kappa = \delta(\mathtt{x}, \mathtt{S}\tau, \Delta)}{\Delta \mid \Gamma \vdash \mathtt{x} : (\kappa.\mathtt{S}\tau, \mathtt{S})} \ (\mathit{V}\text{AR})$$

$$\frac{\mathtt{C} : \forall\overline{\alpha}.\tau \in \Gamma \quad \mathtt{S} = [\overline{\alpha \mapsto \beta}] \quad \overline{\beta}\ \mathit{fresh}}{\Delta \mid \Gamma \vdash \mathtt{C} : (\mathtt{S}\tau, \mathtt{S})} \ (\mathit{C}\text{ONS})$$

$$\frac{\Delta \mid \Gamma, \mathtt{x} : \alpha \vdash \mathtt{e} : (\kappa.\tau, \mathtt{S}) \quad \alpha\ \mathit{fresh}}{\Delta \mid \Gamma \vdash \lambda\mathtt{x}.\mathtt{e} : (\mathtt{S}(\kappa.\alpha \to \tau), \mathtt{S})} \ (\mathit{L}\text{AM})$$

$$\frac{\begin{array}{c}\Delta \mid \Gamma \vdash \mathtt{e}_1 : (\kappa_1.\tau_1, \mathtt{S}_1) \quad \mathtt{S}' = \mathit{unify}(\tau_2 \to \alpha \sim \tau_1) \quad \alpha\ \mathit{fresh} \\ \Delta \mid \mathtt{S}_1\Gamma \vdash \mathtt{e}_2 : (\kappa_2.\tau_2, \mathtt{S}_2) \quad \mathtt{S} = \mathtt{S}' \circ \mathtt{S}_2 \circ \mathtt{S}_1\end{array}}{\Delta \mid \Gamma \vdash (\mathtt{e}_1\ \mathtt{e}_2) : (\mathtt{S}(\kappa_1 \cup \kappa_2.\alpha), \mathtt{S})} \ (\mathit{A}\text{PP})$$

$$\frac{\begin{array}{cc}\Delta, \mathtt{x} \mid \Gamma \vdash \mathtt{e}_1 : (\kappa_1.\tau_1, \mathtt{S}_1) & \overline{\alpha} = \mathtt{ftv}(\kappa.\tau_1) - \mathtt{ftv}(\mathtt{S}_1\Gamma) \\ \Delta \mid \mathtt{S}_1\Gamma, \mathtt{x} : \forall\overline{\alpha}.\kappa_1.\tau_1 \vdash \mathtt{e}_2 : (\kappa_2.\tau_2, \mathtt{S}_2) & \mathtt{S} = \mathtt{S}_2 \circ \mathtt{S}_1\end{array}}{\Delta \mid \Gamma \vdash \mathtt{let\ x\ =\ e}_1\ \mathtt{in}\ \mathtt{e}_2 : (\mathtt{S}(\kappa_1 \cup \kappa_2.\tau_2), \mathtt{S})} \ (\mathit{L}\text{ET})$$

$$\frac{\Delta \mid \Gamma \vdash (\mathtt{p} : \tau, \mathtt{S}_1) \quad \Delta \mid \Gamma \vdash e : (\kappa.\tau', \mathtt{S}_2) \quad \mathtt{S} = \mathtt{S}_1 \circ \mathtt{S}_2}{\Delta \mid \Gamma \vdash \mathtt{p} \to \mathtt{e} : (\mathtt{S}(\kappa.\tau \to \tau'), \mathtt{S})} \ (\mathit{A}\text{LT})$$

$$\frac{\begin{array}{c}\Delta \mid \Gamma \vdash \mathtt{e}' : (\kappa'.\tau', \mathtt{S}') \\ \Delta \mid \mathtt{S}'\Gamma \vdash_{\mathtt{p}} (\mathtt{p}_i \to \mathtt{e}_i) : (\kappa_i.\tau_i' \to \tau_i, \mathtt{S}_i) \quad \mathtt{para\ i} = 1..\mathtt{n} \\ \mathtt{x} = \mathit{top}(\Delta) // \mathtt{nome\ da\ função\ que\ está\ sendo\ definida} \\ \kappa_i^{\mathtt{x}} = \{\mathtt{x} : \tau \in \kappa_i \mid \mathtt{gtc}(\tau) \neq \emptyset\} \\ \{\mathtt{x} : \tau_i' \to \tau_i\} \cup \mathtt{S}'\Gamma \models^{\mathtt{x}} (\kappa_i^x, Si) \\ \mathtt{u} \to \tau = \mathtt{lcg}(\{\mathtt{S}_i(\tau_i' \to \tau_i)\}^{i=1..n}) \\ \overline{\alpha} = \mathtt{gtv}(\mathtt{u}) - \mathtt{rtv}(\tau) \quad \overline{\kappa}\ \mathtt{variáveis\ Skolem\ fresh} \\ \mathtt{S} = \mathit{unifyAll}([\overline{\alpha \mapsto \kappa}](\mathtt{u} \to \tau), \{\mathtt{S}_i(\tau_i' \to \tau_i)\}^{i=1..n})\end{array}}{\Delta \mid \Gamma \vdash \mathtt{case}\ \mathtt{e}'\ \mathtt{of}\ [\mathtt{p}_i \to \mathtt{e}_i]^{i=1..n} : (\mathtt{S}\tau, \mathtt{S})} \ (\mathit{C}\text{ASE})$$

A regra CASE é o ponto principal do algoritmo. Na inferência de tipos para expressões `case`, é primeiramente inferido o tipo do *scrutinee* `e`. Então, o tipo de cada alternativa é inferido (numa ordem textual, mas a ordem não é relevante). Finalmente, se a expressão `case` envolve construtores GADT, o tipo da expressão `case` é aprimorado, usando uma regra de refinamento de tipos separada (já que alternativas `case` não são expressões). Alternativas `case` distintas para o mesmo construtor devem ser unificadas, mas neste trabalho, por questões de simplicidade, consideramos que cada alternativa tem um construtor distinto.

$$\boxed{\Gamma \models^x \kappa, S}$$

$$\frac{}{\Gamma \models^x \emptyset, id}\ \textit{SAT}_\emptyset \qquad \frac{\Gamma \models^x \{x:\tau\}, S' \quad \Gamma \models^x S'\kappa, S}{\Gamma \models^x \{x:\tau\} \cup \kappa, S}\ \textit{SATREC}$$

$$\frac{\overline{S_i} = sat(\{x:\tau\},\Gamma) \quad \tau' = lcg(\{\overline{S_i}\tau\}) \quad S = unify(\tau' \sim \tau)}{\Gamma \models^x \{x:\tau\}, S}\ \textit{SAT}_{-1}$$

Para inferir o tipo de alternativas `case`, é necessário unificar seus construtores de tipo com o tipo inferido para a expressão `e` (*scrutinee*) do `case`, produzindo uma substituição que é usada para instanciar os tipos dos parâmetros, e adicioná-los no contexto de tipo para inferir o tipo do lado direito das alternativas. Para simplificar a regra CASE essa verificação não foi incluída.

Dado um contexto de tipo Γ e, para cada alternativa `i`, um conjunto de restrições de tipo $\kappa_i.\, \tau'_i \rightarrow \tau_i$, o aprimoramento de tipos gera o tipo $\tau' \rightarrow \tau$. Note que apenas funções que tem alternativas com recursão polimórfica geram restrições. Os julgamentos representados por $\models$ computam a substituição que aprimora o tipo usando a função `sat` (κ, Γ), conforme discutido na Seção 2.5.

$$\begin{array}{lcl} \texttt{unifyAll}(\tau, \emptyset) & = & \texttt{id} \\ \texttt{unifyAll}(\tau, (\{\tau'\} \cup \mathbb{T})) & = & unify(\{\tau \sim \tau'\}) \circ \texttt{unifyAll}(\mathbb{T}) \end{array}$$

A sentença de aprimoramento de tipo trabalha da seguinte forma: para cada alternativa i de uma função GADT `x`, define-se $(\kappa_i)^\star_{\mathtt{x}}$ como o conjunto de restrições que mencionam construtores de tipo GADT (no conjunto de restrições o tipo da alternativa na posição i) e define-se $\mathtt{S}_i$ como a substituição de satisfazibilidade para este conjunto de restrições, em um contexto de tipos que contém suposições de tipos correspondentes a todas as alternativas. Então, definimos que $\tau_1 \rightarrow \tau_2$ é o `lcg` de todo $\overline{\mathtt{S}_i\, (\tau'_i \rightarrow \tau_i)}$. Agora "skolemizamos" $\overline{\alpha}$ (i.e. tratando-as como não-unificáveis), o conjunto de variáveis de tipo introduzidas por generalização de tipos dos parâmetros de um GADT. Variáveis de tipo que ocorrem no tipo do retorno de uma função e também na generalização de um parâmetro de tipo de um ADT recursivo não são skolemizadas (por exemplo, quando o tipo `[a]` é obtido como resultado de uma generalização de `[Int]` e `[Bool]`). Os tipos inferidos de alternativas `case` são então unificados com variáveis de tipo não-skolemizadas. As substituições computadas por satisfazibilidade e unificação são aplicadas para o tipo generalizado, que é então retornado. É importante mencionar que o algoritmo é conservador com expressões CASE que não envolvem GADTs, nesse caso nenhuma variável de tipo é skolemizada e os tipos de todas as alternativas são unificados.

4.1.2 Tratamento de Recursão Polimórfica

Na abordagem proposta, é utilizado o algoritmo para verificação da satisfazibilidade de restrições proposto em (RIBEIRO; CAMARAO; FIGUEIREDO, 2013) para tratamento de recursão polimórfica. A proposta é tratar cada alternativa como se fosse uma definição sobrecarregada, visto que as alternativas podem ter tipos distintos. Dessa forma é possível restringir o uso da recursão polimórfica aos casos onde ela é decidível. Este algoritmo garante a terminação sem impor um critério estático, consiste na inclusão de critérios executados a cada passo da chamada recursiva.

É importante ressaltar que o algoritmo proposto por (RIBEIRO; CAMARAO; FIGUEIREDO, 2013) não resolve a indecidibilidade da satisfazibilidade, existem instâncias onde o algoritmo pode rejeitar programas corretos. Esse comportamento pode ocorrer tanto com o uso de um critério estático quanto com o uso do algoritmo proposto por (RIBEIRO; CAMARAO; FIGUEIREDO, 2013), no entanto, no segundo, essa ocorrência é mais rara, e não se conhece efetivamente nenhum caso prático em que ocorra.

Como o tratamento de CS-SAT não é o escopo deste trabalho, detalhes da implementação de (RIBEIRO; CAMARAO; FIGUEIREDO, 2013) não serão abordados, porém o comportamento do algoritmo fica claro nos exemplos apresentados neste capítulo.

4.2 Avaliação da Abordagem Proposta

O algoritmo proposto para inferência de tipos de funções definidas por casamento de padrão sobre construtores de GADTs pode ser sumarizado em quatro etapas, são elas:

1. Inferir separadamente as suposições de tipo cada uma das alternativas de uma função que utiliza GADT, incluindo restrições no tipo de alternativas que envolvem chamadas recursivas.

2. Aprimorar o tipo das alternativas que apresentam recursão polimórfica, por meio da substituição retornada por `sat`. Essa etapa envolve resolver as restrições geradas pelo passo 1, sendo que o tipo da alternativa j é refinado pela substituição dada por $\mathtt{sat}(\kappa_j, \Gamma)$, onde κ_j é um conjunto de restrições inferidas para a alternativa j e Γ contém as suposições de tipo $\{\mathtt{x} : \sigma_i \mid \mathtt{i} = 1, \ldots, \mathtt{n}\}$, onde `x` é o nome que está sendo definido e σ_i é o tipo da equação i na definição de `x` que não é definido recursivamente.

3. Generalizar o tipo das alternativas por meio da anti-unificação (algoritmo `lcg` definido no Capítulo 2), capturando a dependência entre tipos associados ao GADT.

4. Unificar todos os tipos cuja generalização não gerou uma variável de tipo ligada ao GADT.

Com base nesses passos, e nos conceitos vistos anteriormente, no decorrer desta seção serão apresentados alguns exemplos do uso da abordagem proposta na inferência de tipos de funções que envolvem GADTs, além de exemplos de programas que não são aceitos como bem tipados.

4.2.1 Algoritmos Aceitos

Exemplo 1 ***Inferência do tipo da função testT***

A função `testT`, definida a seguir, não possui tipo principal e é rejeitada pelo compilador GHC. Entretanto, é possível fazer com que essa função seja aceita pelo GHC, provendo, para isso, a assinatura $\mathtt{testT} :: \forall \mathtt{a}.\mathtt{T}\ \mathtt{a} \rightarrow \mathtt{a} \rightarrow \mathtt{a}$ ou a assinatura $\mathtt{testT} :: \forall \mathtt{a}.\mathtt{T}\ \mathtt{a} \rightarrow \mathtt{Bool} \rightarrow \mathtt{Bool}$. O tipo dessa última assinatura é inferido para a função `testT` na abordagem proposta, conforme é descrito a seguir.

```
data T a where
  T1 :: Int → T Bool
  T2 :: T a

testT (T1 n) _ = n > 0
testT T2     r = r
```

Primeiramente são geradas as seguintes suposições de tipo para cada uma das alternativas da definição de `testT`:

```
testT :: T Bool → a → Bool
testT :: T b    → c → c
```

Em casos como o desse exemplo, onde não ocorrem chamadas recursivas, nenhuma restrição é gerada, e, portanto, a substituição retornada no passo de verificação de satisfazibilidade de restrições `sat` é identidade. O próximo passo consiste na anti-unificação dos tipos das alternativas que, nesse caso, resulta no tipo:

```
testT :: T b' → c' → d'
```

A generalização obtida nesse processo permite observar a dependência entre os tipos, para identificar quais os tipos possuem ligação, em todas as alternativas, com o GADT. Os tipos que ocorrerem sem essa associação devem ser unificados. Neste caso, temos que `b'` é associado ao GADT, `c'` e `d'` não, portanto os tipos cuja generalização geraram `c'` e `d'` no caso `(a, c)` e `(Bool, c)` devem ser unificados, resultando no seguinte tipo:

```
testT :: ∀a.T a → Bool → Bool
```

Exemplo 2 ***Inferência do tipo da função eval***

Na definição da função `eval`, apresentada no Capítulo 1, e que transcrevemos abaixo, a ocorrência de chamadas recursivas em algumas alternativas envolve resolver recursão polimórfica. Para tratar esses casos, são geradas restrições que serão usadas no processo de refinamento de tipo.

```
data Term a where
  Lit  :: Int → Term Int
  Inc  :: Term Int → Term Int
  IsZ  :: Term Int → Term Bool
  If   :: Term Bool → Term a → Term a → Term a
  Pair :: Term a → Term b → Term (a,b)

eval :: Term a → a
eval (Lit i)      = i
eval (Inc t)      = 1 + eval t
eval (IsZ i)      = 0 == eval i
eval (If b c d)   = if eval b then eval c
                    else eval d
eval (Pair a b)   = (eval a, eval b)
```

Neste caso, os tipos e restrições ineridos para cada uma das alternativas são:

```
(Lit i)         eval :: Term Int → Int
                { }
(Inc t)         eval :: Term Int → Int
                {eval : Term Int → Int}
(IsZ i)         eval :: Term Bool → Bool
                {eval : Term Bool → Bool}
(If l e1 e2)    eval :: Term a → b
                {eval : Term Bool → Bool, eval : Term a → b}
(Pair x y)      eval :: Term (c,d) → (e,f)
                {eval : Term c → e, eval : Term d → f}
```

A alternativa referente ao construtor `If` apresenta chamadas recursivas para todos os argumentos do construtor (`l`, `e1` e `e2`): `l` tem tipo `Term Bool`, gerando a primeira restrição; `e1` e `e2` tem tipo polimórfico `Term a`, porém não existe no corpo da função (nem no contexto) nenhuma declaração que force que o retorno da chamada recursiva seja do mesmo tipo polimórfico `a`, gerando a segunda restrição (`eval : Term a` $\rightarrow$ `b`) . Pelo mesmo motivo, o tipo desta alternativa é inferido como `Term a` $\rightarrow$ `b`. A inferência da alternativa referente ao construtor `Pair` acontece de forma semelhante.

Concluída essa etapa, os tipos das alternativas que apresentam recursão polimórfica são aprimorados por meio da substituição retornada pelo passo `sat`. Ao fim deste processo, serão geradas as seguintes suposições de tipo:

```
(Lit i)        eval :: Term Int → Int
(Inc t)        eval :: Term Int → Int
(IsZ i)        eval :: Term Bool → Bool
(If l e1 e2)   eval :: Term a → a
(Pair x y)     eval :: Term (c,d) → (c,d)
```

O tipo inferido para função `eval` é dado pela anti-unificação do tipo de todas as alternativas, resultando em:

```
eval :: ∀a.Term a → a
```

Neste caso, todos os tipos estão associados ao GADT, o que os caracteriza como tipos que não devem ser unificados.

Exemplo 3 ***Inferência do tipo da função testRep***

No exemplo abaixo (retirado de (JONES et al., 2006)), são definidos dois GADTs. Em `Equal` o construtor `Eq` define que `a` e `b` tem o mesmo tipo. No GADT `Rep`, o construtor `RI` tem tipo concreto e o construtor `RP` pode receber dois tipos distintos como parâmetro, e retorna uma dupla `Rep` (a,b). Em (JONES et al., 2006) o tipo de `testRep` é definido a partir da assinatura de tipos para cada alternativa, pois os tipos de `s1`, `t1`, `s2` e `t2` são rígidos, não sendo possível executar o refinamento de tipos. A abordagem proposta realiza o refinamento de tipos a partir das chamadas recursivas, portanto não é necessária a assinatura.

```
data Equal a b where
     Eq    :: Equal a a

data Rep a where
     RI    :: Rep Int
     RP    :: Rep a → Rep b → Rep (a,b)

testRep (RI RI) = Just Eq
testRep (RP s1 t1) (RP s2 t2)
    = case (testRep s1 s2) of
            Nothing → Nothing
            Just Eq → case (testRep t1 t2) of
                           Nothing → Nothing
                           Just Eq → Just Eq
```

Os tipos e restrições inferidos para cada alternativa são relacionados a seguir:

```
testRep :: Rep Int → Rep Int → Maybe (Equal a' a')    { }
testRep :: Rep (a,b) → Rep (c,d) → Maybe (Equal e e)
           {testeRep : Rep a → Rep c → Maybe (Eq f f),
            testeRep : Rep b → Rep d → Maybe (Eq g g)}
```

Na primeira alternativa, o tipo dos parâmetros é obtido diretamente do tipo do construtor `Maybe` (`Equal a' a'`), onde `a'` é uma variável nova. Na segunda alternativa, são definidas restrições para cada cláusula `case`, e a função tem tipo `Rep (a,b)` para cada parâmetro `e`, assim como a primeira alternativa, retorna um dado `Maybe`. Para todas as alternativas o tipo do retorno é dado por variáveis livres, uma vez que na definição de `Eq` não existe correlação do tipo de `Rep` com o tipo de `Equal`.

O passo de satisfazibilidade de restrições resulta em uma substituição para cada alternativa. Aplicando cada substituição ao tipo da alternativa correspondente, obtêm-se os seguintes tipos:

```
testRep :: Rep Int → Rep Int → Maybe (Equal a' a')
testRep :: Rep (a,b) → Rep (a,b) → Maybe (Equal b' b')
```

Realizando a anti-unificação para os tipos ligados ao GADT e a unificação para tipos ligados a ADTs (neste exemplo `Maybe`), obtem-se o seguinte tipo:

```
testRep :: ∀a.∀b.Rep a → Rep a → Maybe (Equal b b)
```

4.2.2 Algoritmos não Aceitos

Funções com Tipo Recursivo: Em Haskell, tipos algébricos de dados podem ser definidos de forma recursiva, como ocorre com tipos como listas ou árvores. Tipos algébricos são homogêneos, isto é, todos os seus elementos são do mesmo tipo. Assim, uma lista composta por 1, 2 e 3 (representada por `(1:(2:(3:[ ])))` `:: [Int]`) é uma construção válida, mas será reportado um erro caso seja adicionado um caractere nesta lista `(1:(2:(3:('a':[ ]))))`. Os construtores dos tipo algébrico de listas tem os seguintes tipos:

```
(:) :: a → [a] → [a]
[ ] :: [a]
```

Na inferência de tipos para funções que manipulam ADTs, tal como a função `insert` abaixo, definida sobre listas, os tipos das alternativas de definição são unificados (a primeira e a segunda alternativas devem ter o mesmo tipo).

```
insert :: Ord a ⇒ a → [a] → [a]
insert e [ ] = [e]
insert e s@(x:xs) | e < x = e:s
                  | e > x = x : insert e xs
                  | otherwise = s
```

Ao contrário de ADTs, GADTs permitem generalizar o resultado: funções como `eval`, apresentada no Capítulo 1, devem ser aceitas pelo sistema de tipos. No entanto, quando o retorno de funções que usam GADTs envolve tipos de dados recursivos (ADTs), o tipo do retorno deve ser unificado, já que os construtores dos ADTs são homogêneos. A função `proj`, apresentada abaixo, retorna uma lista de termos, utilizando o GADT `Term`, apresentado no Capítulo 1.

```
proj (Lit i)       = [Lit i]
proj (Inc t)       = proj t
proj (IsZ i)       = proj i
proj (If l e1 e2) = proj b ++ proj e1 ++ proj e2
proj (Pair a b)    = proj a ++ proj b
```

Na abordagem proposta neste trabalho, a inferência de tipo para essa função é feita conforme os seguintes passos. Primeiramente, são inferidos os seguintes tipos e restrições para cada uma das alternativas da definição de `proj`:

```
(Lit i)         proj :: Term Int → [Term Int] { }
(Inc t)         proj :: Term Int → a    {proj : Term Int → a}
(IsZ i)         proj :: Term Bool → b   {proj : Term Int → b}
(If l e1 e2)    proj :: Term d → [c]
                {proj : Term Bool → [c], proj : Term d → [c]}
(Pair a b)      proj :: Term (e,f) → [g]
                {proj : Term e → [g], Term f → [g]}
```

Aplicando as substituições obtidas no passo de verificação da satisfazibilidade de restrições, obtêm-se os seguintes tipos:

```
(Lit i)        proj :: Term Int → [Term Int]
(Inc t)        proj :: Term Int → [Term Int]
(IsZ i)        proj :: Term Bool → [Term Int]
(If l e1 e2)   proj :: Term t1 → [Term Int]
(Pair a b)     proj :: Term (t2,t3) → [Term Int]
```

Novamente, a anti-unificação é usada para avaliar os tipos que estão ou não ligados ao GADT, generalizando os tipos ligados ao GADT, exceto nos casos em que esses tipos envolvem construtores de tipos recursivos (ADTs), como ocorre no tipo de `proj`. Os tipos que não são generalizados são unificados, resultando no seguinte tipo para `proj`. Temos então que o tipo de `proj` é:

```
proj :: Term a → [Term Int]
```

Portanto, a função é considerada bem tipada pois foi possível unificar o resultado, que envolve um tipo recursivo. Porém considere a função `proj'`, uma extensão da função `proj` porém adicionando novos construtores ao GADT `Term`:

```
data Term a where
    ...
    LitB    :: Bool → Term Bool
    IsFalse :: Term Bool → Term Bool

proj' (Lit i)      = [Lit i]
proj' (LitB b)     = [Lit b]
proj' (Inc t)      = proj' t
proj' (IsZ i)      = proj' i
proj' (IsFalse b)  = proj' b
proj' (If l e1 e2) = proj' b ++ proj' e1 ++ proj' e2
proj' (Pair a b)   = proj' a ++ proj' b
```

Os tipos e restrições inferidos para essas alternativas são:

```
(Lit i)        proj' :: Term Int → [Term Int]
               { }
(LitB i)       proj' :: Term Bool → [Term Bool]
               { }
(Inc t)        proj' :: Term Int → a
               {proj' : Term Int → a}
(IsZ i)        proj' :: Term Bool → b
               {proj' : Term Int → b}
(IsFalse b)    proj' :: Term Bool → c
               {proj' : Term Bool → c}
(If l e1 e2)   proj' :: Term e → [d]
               {proj' : Term Bool → [d], proj' : Term e → [d]}
(Pair a b)     proj' :: Term (f,g) → [h]
               {proj' : Term f → [h], proj' : Term g → [h]}
```

Pela aplicação das substituições obtidas no passo de verificação da satisfazibilidade de restrições, obtêm-se os seguintes tipos:

```
(Lit i)        proj' :: Term Int → [Term Int]
(LitB b)       proj' :: Term Bool → [Term Bool]
(Inc t)        proj' :: Term Int → [Term Int]
(IsZ i)        proj' :: Term Bool → [Term Int]
(IsFalse b)    proj' :: Term Bool → [Term Bool]
(If l e1 e2)   proj' :: Term t1 → [Term t2]
(Pair a b)     proj' :: Term (t3,t4) → [Term t5]
```

Neste caso não é possível unificar os tipos `Term Int`, `Term Bool`, `Term t2` e `Term t5`, sendo reportado um erro de tipo pelo compilador. Note que a função `proj'` tem como retorno uma lista, que é um ADT, o que torna obrigatório que seus elementos sejam homogêneos. Uma lista de um GADT com tipo polimórfico, tal como `Term a`, vai contra as regras de construção de tipos de dados algébricos. Para tratar estes casos, em nosso algoritmo variáveis livres criadas na inferência de GADTs, mas que ocorrem como argumentos de construtores de tipo de ADTs recursivos, são retiradas da lista de variáveis skolemizadas.

Funções com Diferentes Definições para o Mesmo Construtor: Na definição de um novo tipo de dado, é associado um tipo a cada construtor. Nos ADTs este tipo é inferido, já nos GADTs este tipo é assinado pelo programador. Nos GADTs o parâmetro de tipo pode variar, permitindo que cada construtor tenha um tipo diferente, e a inferência é feita a partir da generalização do tipo das alternativas. Porém, em funções com diferentes alternativas para o mesmo construtor, os tipos dessas alternativas devem ser unificados, como ocorre no caso da função `testMkp` a seguir:

```
data Pair a b where
     MkP :: a → b → Pair a b

testMkp (MkP x True) = x
testMkp (MkP False y)= y
```

cujo tipo será:

```
testMkp :: Pair Bool Bool → Bool
```

Funções cujo Retorno não está Associado ao GADT: Em alguns casos o algoritmo de anti-unificação não consegue capturar a relação entre tipos e alternativas, como por exemplo na função a seguir, apresentada em (STUCKEY; SULZMANN, 2002):

```
data Erk a b where
  I :: Int → Erk Int b
  B :: Bool → Erk a Bool
```

```
fErk (I a) = a + 1          Erk Int b → Int   { }
fErk (B b) = b && True      Erk a Bool → Bool { }
```

A generalização do tipo das alternativas de `fErk` é `Erk a b → c`. Como a variável de tipo `c` não está associada ao GADT, os tipos de retorno das alternativas (`Int` e `Bool`) devem ser unificados, o que não é possível e, portanto, a definição de `fErk` é rejeitada. No entanto, se a função fosse assinada com o tipo `Erk a a → a` esta função poderia ser considerada correta.

5 CONCLUSÃO

Os primeiros trabalhos relacionados a inferência de funções que manipulam GADTs exigiam assinatura de tipos em todos os casos. A primeira abordagem, Wobbly Types, também foi a primeira a ser adicionada como extensão do compilador GHC. Wobbly Types trazia problemas relacionados à propagação do tipo assinado para escopos internos, em expressões aninhadas. Foi então proposto OutsideIn, com a finalidade de corrigir esses problemas. A partir deste ponto, outras abordagens foram propostas com o objetivo de flexibilizar ainda mais o uso de GADTs, permitindo a inferência de tipos sem a necessidade de assinatura: algumas abordagens inferindo um conjunto mais abrangente de programas e outras, um conjunto mais restrito, dependendo das escolhas relacionadas a garantia da propriedade de tipo principal, recursão polimórfica, e outros pontos que serão discutidos nessa seção.

OutsideIn (VYTINIOTIS et al., 2011) propôs uma abordagem segura, com a premissa de inferir tipo apenas expressões com tipo principal, solicitando ao programador informações de tipo nos demais casos. Em P (LIN; SHEARD, 2010) foi proposto um sistema de tipos completo, com a premissa de que obter tipo principal em expressões envolvendo GADTs não é possível na grande maioria dos casos, e então optou por inferir um tipo possivelmente ideal para um conjunto maior de programas, do que garantir a propriedade de tipo principal. Na mesma linha, Chore (CHEN; ERWIG, 2016) busca inferir um conjunto mais abrangente de programas, mesmo que muitas vezes o tipo inferido não seja o principal.

No geral, trabalhos que envolvem inferência de tipos para funções GADT (sem a necessidade de assinatura para um maior número de programas) o fazem a partir de três passos principais: 1) um tipo é atribuído a cada alternativa, com base no tipo assinado para o construtor GADT; 2) uma etapa de refinamento de tipos (também chamada de aprimoramento de tipos), que consiste em capturar a relação do tipo dos elementos com o GADT; 3) por fim, uma etapa de reconciliação de tipos é executada, que consiste num *solver*, que atribuirá um tipo único, podendo ou não ser o tipo principal da expressão. A forma como esses passos são executados varia entre as abordagens, além de questões envolvendo o tratamento de recursão polimórfica, a decisão sobre a inferência de tipos para expressões que não possuem tipo principal, os casos onde deve ser solicitada assinatura de tipos, a necessidade de propagação de tipos, e a forma verificação de erros. Esses tópicos são discutidos a seguir, sem o intuito de classificar as escolhas como certas ou erradas, mas apenas como itens a serem analisados para evolução destas e de novas abordagens.

5.1 O Sistema de Tipos é Incompleto

Diferentes sistemas de tipos tem diferentes critérios para definir quais serão os programas aceitos como bem tipados (*well-typed*) ou não. Um sistema de tipo consistente (*type soundness*), garante que um programa bem tipado é executado sem que ocorram erros em tempo de execução (DAMAS; MILNER, 1982). A consistência de um sistema de tipos é um requisito importante não apenas por questões de formalismo da documentação, como também por questões de formalismo para verificação dos programas. Em linguagens de programação com um sistema de tipos consistente, mesmo que o programador não saiba o tipo que o programa deve ter, ele sabe que o programa tem um tipo e isso dá algumas garantias sobre o comportamento do programa ao executar (LIN, 2010).

Projetar um sistema de tipos de tipagem estática consistente (*todo programa bem tipado não apresenta erros de tipo*) e completo (*todo programa que não apresenta erros, deve ser bem tipado*), embora desejável, é muito difícil na prática. No geral, ao projetar um sistema de tipos, opta-se por um sistema consistente, mas incompleto. Com isso, podem ser rejeitados programas corretos, mas é mantida a segurança, ao invés de aceitar programas incorretos indevidamente.

Desse modo, em um sistema de tipos consistente, todo programa incorreto é considerado mal tipado (*ill-typed*), porém alguns programas corretos podem ser rejeitados, o que pode obrigar que o programador tenha que alterar um programa correto para contornar a incompletude do sistema de tipos. Isso ocorre, por exemplo, com a função `eval` (veja Capítulo 1) utilizando ADTs: como o resultado da função pode ter vários tipos (inteiros, booleanos ou pares), é necessária a definição de um tipo de dado auxiliar que encapsula o resultado de `eval`.

Essa medida alternativa para implementar a função `eval` atende o sistema de tipos de Haskell, porém pode gerar problemas adicionais na execução, como a possibilidade de construir expressões inválidas. Isso ocorre pois a forma como `Term` foi definido é muito tolerante: o tipo dos construtores não é explícito e os erros acabam sendo identificados apenas em tempo de execução.

Como já foi mencionado, esse problema pode ser contornado com o uso de GADTs, que permitem a generalização do resultado da função. Isso só é possível pois nos GADTs o tipo dos construtores é dado de forma explícita, garantindo (na maioria dos casos) a consistência do programa. No entanto, alguns problemas são encontrados na inferência de programas que utilizam GADTs. A maioria destes problemas é resolvido requerendo que o programador forneça a assinatura de tipos para determinadas definições no programa.

5.2 Tipo Principal

A propriedade de tipo principal significa que toda expressão bem tipada tem um tipo mais geral, que representa todos os tipos possíveis para essa expressão. Esta propriedade é importante para o algoritmo de inferência de tipos, pois possibilita definir um único tipo para cada expressão bem tipada.

Porém, devido às restrições de tipo impostas por GADTs, existem casos onde pode haver perda da propriedade de tipo principal, ou seja, não existir um tipo mais geral capaz de representar a expressão. Isso é ilustrado pela função `test` abaixo (exemplo extraído de (SCHRIJVERS et al., 2009)):

```
data T a where
  T1 :: Int → T Bool
  T2 :: T a

test (T1 n) _ = n > 0
test T2     r = r
```

Na primeira alternativa definida para a função `test`, o tipo do retorno `Bool`, inferido para a expressão `n > 0`, está associado à assinatura de tipo do construtor `T1`, que determina que (`T1 n`) tem tipo `T Bool` e `n` tem tipo `Int`. Na segunda alternativa de `test`, não existe nenhuma associação explícita entre o tipo `T a`, construído por `T2`, e o tipo do retorno (tipo de `r`). Neste caso o tipo de retorno deve ser unificado com a primeira alternativa (`Bool`). Considerando que o objetivo no uso de GADTs é possibilitar tipos de retornos distintos, podemos considerar que a relação entre o tipo do GADT e o tipo do retorno está implícito e, neste caso, o retorno teria tipo `a`. Sendo assim, as duas assinaturas abaixo são possíveis inferências para tipo da função `test` (onde nenhuma das alternativas é uma instância da outra):

$$\texttt{test} :: \forall \texttt{a}.\texttt{T a} \rightarrow \texttt{Bool} \rightarrow \texttt{Bool}$$
$$\texttt{test} :: \forall \texttt{a}.\texttt{T a} \rightarrow \texttt{a} \rightarrow \texttt{a}$$

No sistema de tipos de Damas-Milner o tipo principal é obtido pela unificação das alternativas, tendo um único tipo para cada expressão. No entanto, em extensões do sistema de Damas-Milner com suporte a inferência de GADT, o tipo associado ao GADT precisa ser generalizado, perdendo a propriedade de tipo principal em muitos casos. Como nem toda expressão GADT tem tipo principal, pode-se afirmar que o algoritmo de inferência é portanto incompleto (SULZMANN; SCHRIJVERS; STUCKEY, 2008).

A perda da propriedade de tipo principal ocorre quando não é possível definir um único tipo máximo para a expressão, que ocorre com frequência na inferência de GADTs. Sabendo da dificuldade na obtenção do tipo principal de expressões GADT, restringir os programas aceitos aos programas em que é possível obter tipo principal é realmente viável? O tipo principal é realmente importante nesses casos?

Restringir a inferência de tipos ao conjunto de programas em que possível obter tipo principal reduz significativamente o número de programas aceitos sem assinatura de tipos. Por outro lado, aceitando esses programas apenas mediante a assinatura de tipos, traz a garantia do escopo de uso da função esperado pelo programador. Como não é possível garantir principalidade e inferência de tipos para todos os casos, cada abordagem trata essa situação de uma forma diferente, visando atender um desses pontos.

Em (STUCKEY; SULZMANN, 2005), já acreditava-se que a propriedade de tipo principal era muito difícil de se obter em funções envolvendo GADTs. Trabalhos recentes, como (LIN; SHEARD, 2010; CHEN; ERWIG, 2016), optam por inferir um conjunto maior de programas em vez de garantir a propriedade de tipo principal. Em (CHEN; ERWIG, 2016) especificamente, foram levantados pontos interessantes que levaram os autores a optar por inferir o tipo de expressões mesmo que não seja o tipo principal. Para eles, apesar dessas expressões não terem tipo principal, podem ter um tipo mais preciso, que melhor descreve a expressão e restringe os tipos aceitos a tipos que potencialmente não ocasionarão erros.

Por exemplo, a função `flop1` a seguir (a definição do dado `R` pode ser encontrada em 17), tem vários tipos possíveis: `R a` $\rightarrow$ `Int`, `R a` $\rightarrow$ `a` ou `R Int` $\rightarrow$ `Int`.

```
flop1 (RI x) = x
```

Esses diferentes tipos podem ser atribuídos a `flop1` pois o tipo do lado esquerdo de `flop1` deriva de `R a`, devido à definição do tipo GADT, que pode ser refinado para `R Int` pois `flop1` foi definido apenas para o construtor `RI`. Já o lado direito de `flop1` deriva do lado esquerdo, podendo ser tanto `a` quanto `Int`.

Dentre os tipos possíveis, o tipo mais preciso, e que evitaria erros em tempo de execução, é `R Int` $\rightarrow$ `Int`, sendo este o tipo inferido tanto por Chore e P, quanto pelo algoritmo proposto.

Acreditamos, assim como em (LIN; SHEARD, 2010; CHEN; ERWIG, 2016), que inferir o tipo mais preciso seja mais viável no contexto de uso de GADTs, pois do contrário se restringiria muito o número de programas corretos aceitos pelo algoritmo de inferência. Além disso, a inferência desses casos torna o uso de GADTs mais natural para o programador que, caso entenda ser necessário, pode assinar o tipo da função com um tipo diferente do inferido.

5.3 Refinamento de Tipo

Em funções envolvendo GADTs, cada alternativa pode ter um tipo diferente e, portanto, é necessário realizar o refinamento de tipos com base em todas as alternativas disponíveis. O tipo da função `eval`, tendo apenas a primeira alternativa, seria `Term Int` $\rightarrow$ `Int`, de acordo com o tipo assinado para o construtor `Lit`:

```
eval :: Term Int → Int
eval (Lit i)  = i
```

Adicionando-se as próximas duas alternativas para `eval`, o tipo deve ser generalizado para `Term a → a`:

```
eval :: Term a → a
eval (Lit i)  = i
eval (Inc t)  = 1 + eval t
eval (IsZ i)  = 0 == eval i
```

A principal questão envolvendo o refinamento de tipos com GADTs está na impossibilidade de unificar os tipos. As primeiras duas alternativas de `eval` retornam `Int`, e a terceira retorna `Bool` e, apesar de não ser possível unificar estas alternativas, `eval` deve ser uma função bem tipada em um sistema de tipos com suporte a GADTs.

5.4 Recursão Polimórfica

A recursão polimórfica não é um recurso específico do sistema de tipos GADT, porém seu uso é mais comum em funções que manipulam GADTs, se comparado com funções que utilizam ADTs. A função `eval` é um exemplo de função que utiliza GADT e na qual ocorre recursão polimórfica: o parâmetro de `eval` na alternativa que define o construtor `IsZ` tem tipo `Term Bool`, mas a definição chama recursivamente `eval` com tipo `Term Int`. Desta forma, a função `eval` é bem tipada apenas em um sistema de tipos que suporta recursão polimórfica.

A inferência de tipos para expressões envolvendo recursão polimórfica é indecidível, assim como é indecidível avaliar a satisfazibilidade de um conjunto de restrições CS-SAT, fazendo com que a inferência de tipos não termine. Em Haskell, devido à não terminação, e por ser pouco comum o uso, optou-se por tratar casos de recursão polimórfica apenas a partir da assinatura de tipos fornecida pelo programador. No entanto, em funções envolvendo GADTs o uso de recursão polimórfica é recorrente, como já citado, visto que o parâmetro de tipo de um dado GADT assume o tipo de cada construtor, que pode variar.

Diferentes abordagens tratam recursão polimórfica na inferência GADT de diferentes formas, a maioria delas opta por apenas verificar o tipo informado pelo programador (VYTINIOTIS et al., 2011; CHEN; ERWIG, 2016), outras abordagens optam por inferir incluindo um limite de iteração (LIN; SHEARD, 2010). Em nossa abordagem tratamos a recursão polimórfica como chamadas a alternativas sobrecarregadas e utilizamos o algoritmo `sat` proposto em (RIBEIRO; CAMARAO; FIGUEIREDO, 2013).

5.5 Propagação de Restrições de Tipo

Em Haskell (sem considerar extensões), toda expressão tem tipo principal[1]. Na definição de funções envolvendo ADTs apenas, esta propriedade é garantida, pois o parâmetro de tipo não varia e alternativas `case` podem ser unificadas. No entanto, funções envolvendo GADTs não podem ter as alternativas unificadas, visto que o retorno da função varia de acordo com o argumento de tipo GADT. Sem tipo principal, um tipo menos representativo pode ser associado à função indevidamente, impossibilitando sua execução em certos escopos de uso.

Pensando nisso, em Chore foi proposto associar um tipo mais completo às expressões GADT, que neste caso são tipos *choice*. Com isso, todo construtor implementado na função, tem um tipo que é adicionado ao tipo inferido para função.

A obtenção deste tipo completo em funções GADT é possível pois, em abordagens mais recentes, o primeiro passo da inferência envolve atribuir um tipo para cada alternativa (tipo esse que será refinado), e são esses tipos que Chore propaga durante todas as etapas da inferência.

A principal vantagem levantada por (CHEN; ERWIG, 2016), que justifica a propagação de tipos, está no tratamento de erros na aplicação de funções. Segundo os autores, informações de tipo importantes são perdidas quando os tipos são reconciliados para cada expressão. Porém, se por um lado é possível

1 Em alguns casos, bastante específicos, assinaturas de tipo podem levar a perda da propriedade de tipo principal (FAXÉN, 2003).

tratar um volume maior de erros, por outro lado o processo de inferência se torna mais complexo e mais difícil de ser integrado ao sistema de tipos de Haskell. Na regra de aplicação de funções de Chore, por exemplo, é necessário adicionar novos operadores para avaliar se a função é bem tipada. Além disso, as fases da inferência se tornam ilegíveis para o programador, já que o tipo manipulado não é um tipo polimórfico usual.

Assim como em Chore, em nossa abordagem o tipo gerado para as alternativas também poderá ser propagado. Porém, acreditamos que é necessária uma análise mais aprofundada para avaliar os benefícios trazidos versus a complexidade que seria adicionada ao sistema de tipos. Além disso, optamos por não propagar tipos para manter nosso sistema de tipos mais fiel ao sistema de tipos de Damas-Milner e, consequentemente, à linguagem Haskell. Assim como em nossa abordagem, em P a reconciliação dos tipos é feita assim que é inferido o tipo para todas as alternativas.

Acreditamos, no entanto, que esta questão merece uma maior discussão, visto que não existe um estudo completo que prove a maior eficiência de uma ou outra abordagem, avaliando efetivamente quando deve ser feita a reconciliação de tipos.

5.6 Verificação de Erros

Apesar de não ser o escopo deste trabalho, acreditamos ser uma discussão interessante, já que tanto inferir quanto assinar tipos para funções GADT é extremamente difícil. Esta preocupação já foi levantada em Chore, que optou por propagar tipos principalmente para possibilitar o retorno de mensagens mais claras.

Em Chore, o tratamento de erro foi dado principalmente na aplicação de funções, como ocorre em `h2` (cujo código pode ser consultado na Figura 20). Neste exemplo, o tratamento de erro ocorre afim de rejeitar a aplicação de funções ao ser passado como argumento um valor de um tipo não implementado. Situações como essa não tem tratamento em Haskell, pois são considerados erros de implementação. Comparado com funções ADT, seria o equivalente a implementar uma função que manipula lista, sem considerar o construtor de lista vazia:

```
len (x:xs) = 1 + len xs

appLen = len []
```

Como discutido no tópico anterior, a inclusão deste tratamento em Chore envolve adicionar propagação de tipos, que traz complexidade e foge das premissas do sistema de tipos e da linguagem Haskell, sendo que a rejeição de expressões como `appLen` ou `h2` é algo comum para programadores Haskell.

Optamos por escolhas mais conservadoras em nossa abordagem, tornando mais fácil a adequação ao sistema de tipos atual. No entanto, acreditamos que questões relacionadas ao tratamento de erro são importantes, principalmente com relação à clareza das mensagens, mas exige um estudo mais aprofundado.

5.7 Uso de Assinatura de Tipos

O sistema de tipos de Damas-Milner é consistente e garante a inferência de tipos, sem a necessidade de assinatura, para o conjunto de expressões previstas (conforme já mostrado no Capítulo 2). Neste caso, o uso de assinatura de tipos é opcional, e usado apenas para restringir o tipo de expressões.

Extensões do sistema de tipos de Damas-Milner, no entanto, podem exigir informação de tipo para garantir consistência, como é o caso do tratamento de sobrecarga, que em Haskell é implementado a partir de classes de tipo, onde cada instância sobrecarregada do símbolo tem um tipo associado. Da mesma forma, extensões que implementam GADT requerem informação de tipo para aceitação de determinadas funções. O número de expressões que requerem assinatura varia dependendo da abordagem. Em comum entre as abordagens está a dificuldade em delimitar os casos em que o algoritmo de inferência é capaz de capturar o tipo de funções e os casos onde a assinatura de tipo é obrigatória.

Em abordagens que inferem tipo mesmo para expressões sem tipo principal esta questão é ainda mais difícil, adicionado à complexidade na delimitação dos programas onde é possível obter tipo principal, que é uma questão até hoje em discussão. Assim como nos demais trabalhos, ainda não foi possível delimitar o conjunto de programas em que o tipo é inferido e para quais programas é necessária assinatura de tipos. Em trabalhos futuros, almejamos incluir o tratamento de assinatura de tipos, de forma semelhante ao OutsideIn, e avaliar os casos em que a assinatura de tipos é realmente necessária.

5.8 Expressões case com tipos aninhados

Devido à complexidade de tipos de dados generalizados, com ou sem assinatura de tipos, avaliar o tipo de expressões que manipulam GADTs em escopos internos é extremamente difícil.

Na Figura 22, o tipo da expressão é direto (`param1o :: G a b → Int`), visto que o tipo GADT ocorre apenas no escopo mais externo, portanto o retorno do case interno é unificado. No entanto, considerando que `param1o` fosse reescrito incluindo os construtores GADT no segundo nível `case`, a inferência fica ainda mais difícil, visto que a alternativa `True` para `e1` retorna um inteiro (não ligado ao GADT), e o retorno do segundo `case` é um GADT:

```
param1o’ e1 e2 = case e1 of
                   True → 1
                   False → case e2 of
                             G1 e2 → e2 + 1
                             G2 e2 → False
```

E a complexidade aumenta ainda mais, quando outros níveis são adicionados, todos eles com GADTs.

Desde Wobbly Types, esta questão vem sendo discutida. Nesta abordagem havia inclusive uma série de problemas relacionados a propagação de tipos em escopos internos, que foram apenas resolvidos em OutsideIn. OutsideIn, por sua vez, trata esta questão trazendo todas as restrições de tipo para fora do escopo, e então resolvendo. Mais recentemente, em Chore, o tipo de expressões neste formato não são inferidos, e, quando utilizada assinatura de tipos, o tratamento é semelhante à OutsideIn. Na abordagem proposta neste trabalho, esses casos também não são tratados, ficando como trabalho futuro adicionar tratamento tanto na inferência (se possível) quanto mediante à assinatura de tipos.

5.9 Considerações Finais

Neste trabalho é proposto um novo algoritmo de inferência do tipo de funções na presença de GADTs. Esse algoritmo utiliza uma abordagem similar ao tratamento de sobrecarga em mundo fechado para lidar com casos que envolvem recursão polimórfica, que são bastante frequentes em funções envolvendo GADTs. Anti-unificação é usada para capturar a relação entre os tipos das diversas alternativas de uma função, identificando os tipos que devem ser unificados e os tipos que não devem. Em uma análise empírica de uma amostragem de programas, foi possível observar que a abordagem proposta possibilita a inferência de tipos para um conjunto muito menos restrito de funções, mesmo em situações onde é necessário garantir o tipo principal. Uma proposta parcial deste trabalho foi apresentada em (GELAIN et al., 2015).

Almeja-se em trabalhos futuros adicionar o suporte a assinatura de tipos, avalia-se utilizar restrições de igualdade de forma semelhante ao apresentado em OutsideIn; além disso, pretende-se avaliar os casos onde é possível garantir tipo principal, e incluir tratamento para os casos onde tipos GADT ocorrem nos escopos internos de expressões case.

REFERÊNCIAS BIBLIOGRÁFICAS

AUGUSTSSON, L.; PETERSSON, K. Silly Type Families. Gothenburg, Sweden, 1994.

BAARS, A. I.; SWIERSTRA, S. D. Typing Dynamic Typing. In: **7th ACM SIGPLAN International Conference on Functional Programming**. [S.l.]: ACM Press, 2002.

CAMARAO, C.; FIGUEIREDO, L.; VASCONCELLOS, C. Constraint-set Satisfiability for Overloading. In: ACM. **6th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming**. [S.l.], 2004.

CAMARÃO, C.; VASCONCELLOS, C.; FIGUEIREDO, L.; NICOLA, J. Open and closed worlds for overloading: a definition and support for coexistence. **J. UCS**, 2007.

CHEN, S.; ERWIG, M. Principal Type Inference for GADTs. In: **43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages**. New York, NY, USA: ACM, 2016.

CHENEY, J.; HINZE, R. A Llightweight Implementation of Generics and Dynamics. In: CHAKRAVARTY, M. M. (Ed.). **2002 ACM SIGPLAN Workshop on Haskell (Haskell '02)**. New York, NY, USA: ACM, 2002.

CHENEY, J.; HINZE, R. **First-Class Phantom Types**. [S.l.], 2003.

DAMAS, L.; MILNER, R. Principal Type-Schemes for Functional Programs. In: **9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages**. New York, NY, USA: ACM, 1982.

FAXÉN, K.-F. Haskell and Principal Types. In: **Proceedings of the 2003 ACM SIGPLAN Workshop on Haskell**. New York, NY, USA: ACM, 2003. (Haskell '03).

GELAIN, A.; VASCONCELLOS, C.; CAMARAO, C.; RIBEIRO, R. Type Inference for GADTs and Anti-Unification. **XIX SBLP**, Belo Horizonte, MG, Brasil, 2015.

HENGLEIN, F. Type Inference with Polymorphic Recursion. **ACM Trans. Program. Lang. Syst.**, ACM, New York, NY, USA, 1993.

JAHAMA, S.; KFOURY, A. J. **A General Theory of Semi-Unification**. Boston, MA, USA, 1993.

JONES, M. Type Classes with Functional Dependencies. In: **Programming Languages and Systems**. [S.l.]: Springer Berlin Heidelberg, 2000.

JONES, S. P.; VYTINIOTIS, D.; WEIRICH, S.; WASHBURN, G. Simple Unification-based Type Inference for GADTs. **ICFP**, ACM, New York, NY, USA, 2006.

JONES, S. P.; WASHBURN, G.; WEIRICH, S. **Wobbly Types: Type Inference for Generalised Algebraic Data Types**. Philadelphia, Pennsylvania, 2004.

KFOURY, A. J.; TIURYN, J.; URZYCZYN, P. Type Reconstruction in the Presence of Polymorphic Recursion. **ACM Trans. Program. Lang. Syst.**, ACM, New York, NY, USA, 1993.

LIN, C.-K. **Practical Type Inference for the Gadt Type System**. Tese (Doutorado), Portland, OR, USA, 2010.

LIN, C. kai; SHEARD, T. Pointwise Generalized Algebraic Data Types. In: **5th ACM SIGPLAN Workshop on Types in Language Design and Implementation**. New York, NY, USA: ACM, 2010.

MARTELLI, A.; MONTANARI, U. An Efficient Unification Algorithm. **ACM Trans. Program. Lang. Syst.**, ACM, New York, NY, USA, 1982.

MYCROFT, A. Polymorphic Type Schemes and Recursive Definitions. In: **6th Colloquium on International Symposium on Programming**. [S.l.]: Springer-Verlag, 1984.

OKASAKI, C. **Purely Functional Data Structures**. New York, NY, USA: Cambridge University Press, 1998.

PATERSON, M. S.; WEGMAN, M. N. Linear Unification. In: **Eighth Annual ACM Symposium on Theory of Computing**. New York, NY, USA: ACM, 1976.

PLOTKIN, G. D. A Note on Inductive Generalization. **Machine Intelligence**, 1970.

POTTIER, F.; RÉGIS-GIANAS, Y. Stratified Type Inference for Generalized Algebraic Data Types. In: **33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages**. New York, NY, USA: ACM, 2006.

RIBEIRO, R.; CAMARAO, C.; FIGUEIREDO, L. Terminating Constraint Set Satisfiability and Simplification Algorithms for Context-dependent Overloading. **Journal of the Brazilian Computer Society**, 2013.

SCHRIJVERS, T.; JONES, S. P.; SULZMANN, M.; VYTINIOTIS, D. Complete and Decidable Type Inference for GADTs. In: **14th ACM SIGPLAN International Conference on Functional Programming**. New York, NY, USA: ACM, 2009.

STUCKEY, P.; SULZMANN, M. A Theory of Overloading. In: **7 th ACM International Conference on Functional Programming**. [S.l.]: ACM Press, 2002.

STUCKEY, P. J.; SULZMANN, M. Type Inference for Guarded Recursive Data Types. **Computing Research Repository (CoRR)**, 2005.

SULZMANN, M.; SCHRIJVERS, T.; STUCKEY, P. J. **Type Inference for GADTs via Herbrand Constraint abduction**. [S.l.], 2008.

TEAM, T. G. **The Glorious Glasgow Haskell Compilation System User's Guide, Version 6.4**. [S.l.]: 6.4, 2005.

TEAM, T. G. **The Glorious Glasgow Haskell Compilation System User's Guide, Version 7.10.1**. [S.l.]: 7.10.1, 2015.

VYTINIOTIS, D.; JONES, S. P.; SCHRIJVERS, T.; SUZMANN, M. Outsidein(x) Modular Type Inference with Local Assumptions. **J. Funct. Program.**, Cambridge University Press, New York, NY, USA, 2011.

XI, H.; CHEN, C.; CHEN, G. Guarded Recursive Datatype Constructors. In: **30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages**. New York, NY, USA: ACM, 2003.

Anexos

ANEXO A – ALGORITMOS UTILIZADOS NOS TESTES

```
data Term a where
  Lit  :: Int → Term Int
  Inc  :: Term Int → Term Int
  IsZ  :: Term Int → Term Bool
  If   :: Term Bool → Term a → Term a → Term a
  Pair :: Term a → Term b → Term (a,b)
  Fst  :: Term (a,b) → Term a
  Snd  :: Term (a,b) → Term b

eval :: Term a → a
eval e = case e of
           (Lit i)     → i
           (Inc t)     → eval t + 1
           (IsZ t)     → eval t == 0
           (If b t e)→ if eval b then eval t
                        else eval e
           (Pair a b)→ (eval a, eval b)
           (Fst t)     → fst (eval t)
           (Snd t)     → snd (eval t)
```

Figura 10: Função para avaliação de expressões, com chamadas recursivas.

```
data T a where
  T1 :: Int → T Bool
  T2 :: T a

testT (T1 n) _ = n > 0
testT T2     r = r
```

Figura 11: Função sem tipo principal.

```
data Equal a b where
     Eq   :: Equal a a

data Rep a where
     RI   :: Rep Int
     RP   :: Rep a → Rep b → Rep (a,b)


testRep (RI RI) = Just Eq
testRep (RP s1 t1) (RP s2 t2)
   = case (testRep s1s2) of
         Nothing → Nothing
         Just Eq → case (testRep s1 s2) of
                    Nothing → Nothing
                    Just Eq → Just Eq
```

Figura 12: Expressão com cases aninhados.

```
proj (Lit i)      = [Lit i]
proj (Inc t)      = proj t
proj (IsZ i)      = proj i
proj (If l e1 e2) = proj b ++ proj e1 ++ proj e2
proj (Pair a b)   = proj a ++ proj b
```

Figura 13: Função com tipo recursivo, onde o resultado das alternativas pode ser unificado.

```
proj’ (Lit i)      = [Lit i]
proj’ (LitB b)     = [Lit b]
proj’ (Inc t)      = proj’ t
proj’ (IsZ i)      = proj’ i
proj’ (IsFalse b)  = proj’ b
proj’ (If l e1 e2) = proj’ b ++ proj’ e1 ++ proj’ e2
proj’ (Pair a b)   = proj’ a ++ proj’ b
```

Figura 14: Função com tipo recursivo, onde não é possível unificar o tipo das alternativas.

```
data Pair a b where
     MkP :: a → b → Pair a b

testMkp (MkP x True) = x
testMkp (MkP False y)= y
```

Figura 15: Função com diferentes definições para o mesmo construtor.

```
data Erk a b where
  I :: Int → Erk Int b
  B :: Bool → Erk a Bool

fErk (I a) = a + 1
fErk (B b) = b && True
```

Figura 16: Função cujo retorno não está associado ao GADT.

```
data R a where
  RI :: Int → R Int
  RB :: Bool → R Bool
  BC :: Char → R Char

flop1 (RI x) = x
```

Figura 17: Função com tipo preciso: `flop1` não tem tipo principal, sendo válidos os tipos `R a → a`, `R a → Int` e `R Int → Int`, porém o último é considerado mais preciso por ser menos geral e evitar erros (LIN; SHEARD, 2010; VYTINIOTIS et al., 2011).

```
flop2 (e) = case e of
              RI x → x
              RB x → x
```

Figura 18: Função com tipo generalizado: os tipos das alternativas de `flop2` variam de acordo com tipo dos construtores do GADT.

```
h1 = flop2 (RI 1)
```

Figura 19: Aplicação da função `flop2` para uma alternativa válida.

```
h2 = flop2 (RC 'a')
```

Figura 20: Aplicação da função `flop2` para uma alternativa cujo construtor não foi implementado.

```
cross (RI x) = even x
cross (RB x) = 1
```

Figura 21: Função cujo tipo da alternativa é ligado ao GADT, mas o tipo do retorno não. Novamente ocorreria o tipo `R a → b`, que não é válido.

```
data G a b where
  G1 :: a → G Int a
  G2 :: a → G Bool a

param1o e = case e of
              G1 i → i + 3
              G2 b → case b of
                       True → 4
                       False → 7
```

Figura 22: Função com expressões case aninhadas, o primeiro nível `case` está ligado ao GADT e o segundo nível não está.